USENIX Association

Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation

October 8–10, 2018
Carlsbad, CA, USA

Symposium Organizers

Program Co-Chair

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*

Geoff Voelker, *University of California, San Diego*

Program Committee

Rachit Agarwal, *Cornell University*

Marcos K. Aguilera, *VMware Research*

Mahesh Balakrishnan, *Yale University/Facebook*

Ranjita Bhagwan, *Microsoft Research India*

Edouard Bugnion, *École Polytechnique Fédérale de Lausanne (EPFL)*

Miguel Castro, *Microsoft Research Cambridge*

Kang Chen, *Tsinghua University*

Vijay Chidambaram, The *University of Texas at Austin*

Landon Cox, *Duke University*

Angela Demke Brown, *University of Toronto*

Sandhya Dwarkadas, *University of Rochester*

Sasha Fedorova, *University of British Columbia*

Roxana Geambasu, *Columbia University*

Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

Haryadi Gunawi, *University of Chicago*

Andreas Haeberlen, *University of Pennsylvania*

Jon Howell, *VMware Research*

Rebecca Isaacs, *Twitter*

Michael Isard, *Google*

Frans Kaashoek, *Massachusetts Institute of Technology*

Manos Kapritsos, *University of Michigan*

Taesoo Kim, *Georgia Institute of Technology*

Sam King, *University of California, Davis*

Christoforos Kozyrakis, *Stanford University*

Jinyang Li, *New York University*

Wyatt Lloyd, *Princeton University*

Jay Lorch, *Microsoft Research*

Richard Mortier, *University of Cambridge*

Gilles Muller, *Inria*

KyoungSoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*

Raluca Popa, *University of California, Berkeley*

Don Porter, *The University of North Carolina at Chapel Hill*

Justine Sherry, *Carnegie Mellon University*

Liuba Shrira, *Brandeis University*

Ryan Stutsman, *University of Utah*

Steve Swanson, *University of California, San Diego*

Michael Swift, *University of Wisconsin—Madison*

Dan Tsafrir, *VMware Research and Technion—Israel Institute of Technology*

Rashmi Vinayak, *Carnegie Mellon University*

Xi Wang, *University of Washington*

Andrew Warfield, *Amazon*

Roger Wattenhofer, *ETH Zurich*

Hakim Weatherspoon, *Cornell University*

Ming Wu, *Microsoft Research*

Yubin Xia, *Shanghai Jiao Tong University*

Ding Yuan, *University of Toronto*

Matei Zaharia, *Stanford University*

Irene Zhang, *Microsoft Research*

Yiying Zhang, *Purdue University*

Poster Session Co-Chairs

Vijay Chidambaram, *The University of Texas at Austin*

Yiying Zhang, *Purdue University*

Steering Committee

Brad Chen, *Google*

Jason Flinn, *University of Michigan*

Casey Henderson, *USENIX Association*

Kimberly Keeton, *Hewlett Packard Labs*

Hank Levy, *University of Washington*

James Mickens, *Harvard University*

Brian Noble, *University of Michigan*

Timothy Roscoe, *ETH Zurich*

Margo Seltzer, *University of British Columbia*

Amin Vahdat, *Google and University of California, San Diego*

External Reviewers

Remzi Arpaci-Dusseau
Anish Athalye
Jonathan Behrens
Gino Brunner
Tej Chajed
Nishanth Chandran
Haibo Chen
Charlie Curtsinger
Cody Cutler
Manuel Eichelberger
Jon Gjengset
Michael Gleicher
Joseph Gonzalez
Matthew Hicks
Chris Hodsdon
Atalay Mert İleri
Pankaj Khanchandani
David Lion
Haonan Lu
Yu Luo
Tony Mason
Darya Melnyk
Ellis Michael
Robert Morris
Mihir Nanavati
Luke Nelson
Khiem Ngo
Amy Ousterhout
Pál András Papp
Ali Razeen
Xiang Ren
Oliver Richter
Kirk Rodrigues
Edo Roth
Brian Sandler
Malte Schwarzkopf
Igor Smolyar
Zhenyu Song
Theano Stavrinos
Julian Steger
Simon Tanner
Yuyi Wang
Michael Wei
Tian Yang
Idan Yaniv
Yongle Zhang
Xu Zhao
Aviad Zuck

Message from the OSDI '18 Program Co-Chairs

Dear colleagues,

Welcome to the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), held in Carlsbad, CA, USA! This year's technical program matches OSDI '16 in including 47 exceptionally strong papers; these papers represent the many strengths of our community and cover a wide range of topics, including file and storage systems, networking, scheduling, security, formal verification of systems, graph processing, system support for machine learning, programming languages, fault-tolerance and reliability, debugging, and, of course, operating systems design and implementation.

OSDI '18 received 257 paper submissions, which the program committee reviewed in multiple rounds. Our program committee consisted of 49 reviewers with a mixture of academic and industrial research and practical experience. The PC was divided into 24 "light" and 25 "heavy" members. All papers received three reviews in the first round; based on those reviews, 122 papers were selected to proceed to the second round. Second round papers received a minimum of two additional reviews from heavy PC members. For a small number of papers, where opinions were divided or where a paper was particularly specialized, we solicited additional expert reviews. In total, the PC and external reviewers wrote over 2^10 reviews.

As in previous OSDI review cycles, this year's process included a response period in which authors could answer reviewer questions and address factual errors in the initial reviews. Authors of 191 papers submitted a response. Responses had a measurable impact on both our online and in-person discussions, and the author responses and ensuing online discussion influenced some PC members to adjust their reviews and reconsider their ratings. Overall, we believe author responses helped improve the quality of the selected program.

After more than a week of online discussion across the full PC, we picked 83 papers for the heavy PC members to discuss at a 1.5-day PC meeting held at the University of Wisconsin in Madison, WI, USA. Almost all heavy PC members were able to attend in person, with just one person calling in remotely. As PC chairs, we strove to ensure that all the discussed papers received full and fair consideration, coming to a consensus agreement in most cases. Papers were placed into high-level categories according to their main topic so that similar papers could be discussed together at the PC meeting. All discussed papers received a summary of the PC discussion written by a heavy PC member. In the end, the PC selected 47 papers for presentation at the conference, resulting in an 18% acceptance rate. Each of the accepted papers was allocated an additional two pages and shepherded by a member of the heavy PC to help the authors address the reviewers' comments in their camera-ready versions.

After finalizing the program, we created a separate committee to decide the Jay Lepreau Best Paper Awards composed of PC members with no conflicts with the papers under consideration. PC members could nominate papers for these awards in their reviews or directly to us. We selected six papers with at least two votes for best paper as candidates for the award. After reading the nominated papers and considering the reviews from the full PC, the awards committee agreed on three Jay Lepreau Best Paper Awards.

As PC co-chairs, we stand on the shoulders of so many who did a tremendous amount of hard work to make OSDI '18 a success. First, we thank the authors of all submitted papers for choosing to send their work to OSDI. Thanks also to the program committee for their hard work in reviewing and discussing the submissions and in shepherding the accepted papers. We particularly thank Yiying Zhang and Vijay Chidambaram for organizing an extensive poster session of more than 83 posters to be presented across two evenings. We are also grateful to the external reviewers who provided additional perspectives. We thank the USENIX staff, who have been fundamental in organizing OSDI '18. Finally, OSDI wouldn't be what it is without our attendees—thank you for listening to our speakers, asking challenging and insightful questions, sharing your ideas with others, and networking with one another in the hallways!

We hope you will find OSDI '18 interesting, educational, and inspiring!

Andrea Arpaci-Dusseau, *University of Wisconsin-Madison*
Geoff Voelker, *University of California, San Diego*
OSDI '18 Program Co-Chairs

OSDI '18:
13th USENIX Symposium on
Operating Systems Design and Implementation

October 8–10, 2018
Carlsbad, CA, USA

Understanding Failures

Capturing and Enhancing *In Situ* System Observability for Failure Detection 1
Peng Huang, *Johns Hopkins University;* Chuanxiong Guo, *ByteDance Inc.;* Jacob R. Lorch and Lidong Zhou, *Microsoft Research;* Yingnong Dang, *Microsoft*

REPT: Reverse Debugging of Failures in Deployed Software 17
Weidong Cui and Xinyang Ge, *Microsoft Research Redmond;* Baris Kasikci, *University of Michigan;* Ben Niu, *Microsoft Research Redmond;* Upamanyu Sharma, *University of Michigan;* Ruoyu Wang, *Arizona State University;* Insu Yun, *Georgia Institute of Technology*

Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing 33
Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, and Pandian Raju, *University of Texas at Austin;* Vijay Chidambaram, *University of Texas at Austin and VMware Research*

An Analysis of Network-Partitioning Failures in Cloud Systems 51
Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany, *University of Waterloo*

Operating Systems

LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation 69
Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang, *Purdue University*

The benefits and costs of writing a POSIX kernel in a high-level language 89
Cody Cutler, M. Frans Kaashoek, and Robert T. Morris, *MIT CSAIL*

Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS 107
Ahmed Khawaja, Joshua Landgraf, and Rohith Prakash, *UT Austin;* Michael Wei and Eric Schkufza, *VMware Research Group;* Christopher J. Rossbach, *UT Austin and VMware Research Group*

Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing 129
Kiwan Maeng and Brandon Lucia, *Carnegie Mellon University*

Scheduling

Arachne: Core-Aware Thread Management 145
Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout, *Stanford University*

Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models . . .161
Suli Yang, *Ant Financial Services Group;* Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *UW-Madison*

***μ*Tune: Auto-Tuned Threading for OLDI Microservices 177**
Akshitha Sriraman and Thomas F. Wenisch, *University of Michigan*

RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor 195
Daniel S. Berger and Benjamin Berg, *Carnegie Mellon University;* Timothy Zhu, *Pennsylvania State University;* Siddhartha Sen, *Microsoft Research;* Mor Harchol-Balter, *Carnegie Mellon University*

(continued on next page)

Data

Noria: dynamic, partially-stateful data-flow for high-performance web applications 213
Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, and Lara Timbó Araújo, *MIT CSAIL;* Martin Ek, *Norwegian University of Science and Technology;* Eddie Kohler, *Harvard University;* M. Frans Kaashoek and Robert Morris, *MIT CSAIL*

Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! 233
Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen, *Shanghai Jiao Tong University*

Dynamic Query Re-Planning using QOOP 253
Kshiteej Mahajan, *UW-Madison;* Mosharaf Chowdhury, *U. Michigan;* Aditya Akella and Shuchi Chawla, *UW-Madison*

Focus: Querying Large Video Datasets with Low Latency and Low Cost 269
Kevin Hsieh, *Carnegie Mellon University;* Ganesh Ananthanarayanan and Peter Bodik, *Microsoft;* Shivaram Venkataraman, *Microsoft / UW-Madison;* Paramvir Bahl and Matthai Philipose, *Microsoft;* Phillip B. Gibbons, *Carnegie Mellon University;* Onur Mutlu, *ETH Zurich*

Verification

Nickel: A Framework for Design and Verification of Information Flow Control Systems 287
Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang, *University of Washington*

Verifying concurrent software using movers in CSPEC 307
Tej Chajed and Frans Kaashoek, *MIT CSAIL;* Butler Lampson, *Microsoft;* Nickolai Zeldovich, *MIT CSAIL*

Proving confidentiality in a file system using DISKSEC 323
Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich, *MIT CSAIL*

Proving the correct execution of concurrent services in zero-knowledge 339
Srinath Setty, *Microsoft Research;* Sebastian Angel, *University of Pennsylvania;* Trinabh Gupta, *Microsoft Research and UCSB;* Jonathan Lee, *Microsoft Research*

Reliability

The FuzzyLog: A Partially Ordered Shared Log 357
Joshua Lockerman, *Yale University;* Jose M. Faleiro, *UC Berkeley;* Juno Kim, *UC San Diego;* Soham Sankaran, *Cornell University;* Daniel J Abadi, *University of Maryland, College Park;* James Aspnes, *Yale University;* Siddhartha Sen, *Microsoft Research;* Mahesh Balakrishnan, *Yale University / Facebook*

Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently 373
Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, and Yee Jiun Song, *Facebook;* Tianyin Xu, *Facebook and University of Illinois at Urbana-Champaign*

Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems 391
Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin - Madison*

Taming Performance Variability 409
Aleksander Maricq and Dmitry Duplyakin, *University of Utah;* Ivo Jimenez and Carlos Maltzahn, *University of California Santa Cruz;* Ryan Stutsman and Robert Ricci, *University of Utah*

(continued on next page)

File Systems

Pocket: Elastic Ephemeral Storage for Serverless Analytics 427
Ana Klimovic and Yawen Wang, *Stanford University;* Patrick Stuedi, Animesh Trivedi, and Jonas Pfefferle, *IBM Research;* Christos Kozyrakis, *Stanford University*

Sharding the Shards: Managing Datastore Locality at Scale with Akkio 445
Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, and David Nagle, *Facebook;* Michael Stumm, *University of Toronto*

Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. 461
Pengfei Zuo, Yu Hua, and Jie Wu, *Huazhong University of Science and Technology*

FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. 477
Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, and Changlim Lee, *Yonsei University;* Mohammad Alian, *UIUC;* Myoungjun Chun, *Seoul National University;* Mahmut Taylan Kandemir, *Penn State University;* Nam Sung Kim, *UIUC;* Jihong Kim, *Seoul National University;* Myoungsoo Jung, *Yonsei University*

Debugging

Orca: Differential Bug Localization in Large-Scale Services 493
Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip, *Microsoft Research India*

Differential Energy Profiling: Energy Optimization via Diffing Similar Apps. 511
Abhilash Jindal and Y. Charlie Hu, *Purdue University and Mobile Enerlytics, LLC*

wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events 527
Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang, *The Ohio State University*

Sledgehammer: Cluster-Fueled Debugging 545
Andi Quinn, Jason Flinn, and Michael Cafarella, *University of Michigan*

Machine Learning

Ray: A Distributed Framework for Emerging AI Applications. 561
Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica, *UC Berkeley*

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning 579
Tianqi Chen and Thierry Moreau, *University of Washington;* Ziheng Jiang, *University of Washington, AWS;* Lianmin Zheng, *Shanghai Jiao Tong University;* Eddie Yan, Haichen Shen, and Meghan Cowan, *University of Washington;* Leyuan Wang, *UC Davis, AWS;* Yuwei Hu, *Cornell;* Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy, *University of Washington*

Gandiva: Introspective Cluster Scheduling for Deep Learning. 595
Wencong Xiao, *Beihang University & Microsoft Research;* Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, and Nipun Kwatra, *Microsoft Research;* Zhenhua Han, *The University of Hong Kong & Microsoft Research;* Pratyush Patel, *Microsoft Research;* Xuan Peng, *Huazhong University of Science and Technology & Microsoft Research;* Hanyu Zhao, *Peking University & Microsoft Research;* Quanlu Zhang, Fan Yang, and Lidong Zhou, *Microsoft Research*

Pretzel: Opening the Black Box of Machine Learning Prediction Serving Systems 611
Yunseong Lee, *Seoul National University;* Alberto Scolari, *Politecnico di Milano;* Byung-Gon Chun, *Seoul National University;* Marco Domenico Santambrogio, *Politecnico di Milano;* Markus Weimer and Matteo Interlandi, *Microsoft*

(continued on next page)

Networking

Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage 627
Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman, *University of Utah*

Neural Adaptive Content-aware Internet Video Delivery. 645
Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han, *KAIST*

Floem: A Programming System for NIC-Accelerated Network Applications 663
Phitchaya Mangpo Phothilimthana, *University of California, Berkeley;* Ming Liu and Antoine Kaufmann, *University of Washington;* Simon Peter, *The University of Texas at Austin;* Rastislav Bodik and Thomas Anderson, *University of Washington*

Security

Graviton: Trusted Execution Environments on GPUs 681
Stavros Volos and Kapil Vaswani, *Microsoft Research;* Rodrigo Bruno, *INESC-ID / IST, University of Lisbon*

ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks 697
Radhesh Krishnan Konoth, *Vrije Universiteit Amsterdam;* Marco Oliverio, *University of Calabria/Vrije Universiteit Amsterdam;* Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi, *Vrije Universiteit Amsterdam*

Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. 711
David Lazar, Yossi Gilad, and Nickolai Zeldovich, *MIT CSAIL*

Obladi: Oblivious Serializable Transactions in the Cloud 727
Natacha Crooks, *The University of Texas at Austin;* Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi, *Cornell University*

Graphs and Data

ASAP: Fast, Approximate Graph Pattern Mining at Scale 745
Anand Padmanabha Iyer, *UC Berkeley;* Zaoxing Liu and Xin Jin, *Johns Hopkins University;* Shivaram Venkataraman, *Microsoft Research / University of Wisconsin;* Vladimir Braverman, *Johns Hopkins University;* Ion Stoica, *UC Berkeley*

RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine 763
Kai Wang, *UCLA;* Zhiqiang Zuo, *Nanjing University;* John Thorpe, *UCLA;* Tien Quang Nguyen, *Facebook;* Guoqing Harry Xu, *UCLA*

Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows 783
Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, and Desislava Dimitrova, *ETH Zurich;* Matthew Forshaw, *Newcastle University;* Timothy Roscoe, *ETH Zurich*

Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data 799
Gregory Essertel, Ruby Tahboub, and James Decker, *Purdue University;* Kevin Brown and Kunle Olukotun, *Stanford University;* Tiark Rompf, *Purdue University*

Capturing and Enhancing *In Situ* System Observability for Failure Detection

Peng Huang
Johns Hopkins University

Chuanxiong Guo
ByteDance Inc.

Jacob R. Lorch
Lidong Zhou
Microsoft Research

Yingnong Dang
Microsoft

Abstract

Real-world distributed systems suffer unavailability due to various types of failure. But, despite enormous effort, many failures, especially gray failures, still escape detection. In this paper, we argue that the missing piece in failure detection is detecting what the requesters of a failing component see. This insight leads us to the design and implementation of Panorama, a system designed to enhance *system observability* by taking advantage of the interactions between a system's components. By providing a systematic channel and analysis tool, Panorama turns a component into a logical observer so that it not only handles errors, but also *reports* them. Furthermore, Panorama incorporates techniques for making such observations even when indirection exists between components. Panorama can easily integrate with popular distributed systems and detect all 15 *real-world* gray failures that we reproduced in less than 7 s, whereas existing approaches detect only one of them in under 300 s.

1 Introduction

Modern cloud systems frequently involve numerous components and massive complexity, so failures are common in production environments [17, 18, 22]. Detecting failures reliably and rapidly is thus critical to achieving high availability. While the problem of failure detection has been extensively studied [8, 13, 14, 20, 24, 29, 33, 34, 47], it remains challenging for practitioners. Indeed, system complexity often makes it hard to answer the core question of *what constitutes a failure*.

A simple answer, as used by most existing detection mechanisms, is to define failure as complete stoppage (crash failure). But, failures in production systems can be obscure and complex, in part because many simple failures can be eliminated through testing [49] or gradual roll-out. A component in production may experience gray failure [30], a failure whose manifestation is subtle and difficult to detect. For example, a critical thread of a process might get stuck while its other threads including a failure detector keep running. Or, a component might experience limplock [19], random packet loss [26], fail-slow hardware [11, 25], silent hanging, or state corruption. Such complex failures are the culprits of many real-world production service outages [1, 3, 4, 6, 10, 23, 30, 36, 38].

As an example, ZooKeeper [31] is a widely-used system that provides highly reliable distributed coordination. The system is designed to tolerate leader or follower crashes. Nevertheless, in one production deployment [39], an entire cluster went into a near-freeze status (i.e., clients were unable to write data) even though the leader was still actively exchanging heartbeat messages with its followers. That incident was triggered by a transient network issue in the leader and a software defect that performs blocking I/Os in a critical section.

Therefore, practitioners suggest that failure detection should evolve to monitor *multi-dimensional* signals of a system, aka *vital signs* [30, 37, 44]. But, defining signals that represent the health of a system can be tricky. They can be incomplete or too excessive to reason about. Setting accurate thresholds for these signals is also an art. They may be too low to prevent overreacting to benign faults, or too high to reliably detect failures. For example, an impactful service outage in AWS was due to a latent memory leak, which caused the system to get stuck when serving requests and eventually led to a cascading outage [10]. Interestingly, there was a monitor for system memory consumption, but it triggered no alarm because of "the difficulty in setting accurate alarms for a dynamic system" [10]. These monitoring challenges are further aggravated in a multi-tenant environment where both the system and workloads are constantly changing [44].

In this paper, we advocate detecting complex production failures by enhancing *observability* (a measure of how well components' internal states can be inferred from their external interactions [32]). While defining the absolute health or failure of a system in isolation is tricky,

```
void syncWithLeader(long newLeaderZxid) {
  QuorumPacket qp = new QuorumPacket();
  readPacket(qp);
  try {
    if (qp.getType() == Leader.SNAP) {
      deserializeSnapshot(leaderIs);
      String sig = leaderIs.read("signature");
      if (!sig.equals("BenWasHere"))
        throw new IOException("Bad signature");
    } else {
      LOG.error("Unexpected leader packet.");
      System.exit(13);
    }
  } catch (IOException e) {
    LOG.warn("Exception sync with leader", e);
    sock.close();
  }
}
```

Listing 1: A follower requesting a snapshot from the leader tries to *handle* or *log* errors but it does not *report* errors.

modern distributed systems consist of many highly interactive components across layers. So, when a component becomes unhealthy, the issue is likely observable through its effects on the *execution* of some, if not all, other components. For example, in the previous ZooKeeper incident, even though the simple heartbeat detectors did not detect the partial failure, the Cassandra process experienced many request time-outs that caused its own unserved requests to rapidly accumulate. Followers that requested snapshots from the leader also encountered exceptions and could not continue. Thus, errors encountered in the execution path of interactive components enhance the observability of complex failures.

Even though an interactive component (a *requester*) is well-placed to observe issues of another component (a *provider*) when it experiences errors, such a requester is often designed to **handle** that error but not **report** it (e.g., Listing 1). For example, the requester may release a resource, retry a few times, reset its state, use a cached result (i.e., be fail-static), or exit. This tendency to prioritize error handling over error reporting is possibly due to the modularity principle of "separation of concern" [41, 42], which suggests that components should hide as much information as they can and that failure detection and recovery should be each component's own job. Even if a component has incentive to report, it may not have a convenient systematic mechanism to do so. It can write errors in its own logs to be collected and aggregated by a central service, as is done in current practice. The correlation, however, usually happens in an offline troubleshooting phase, which is too late.

We present Panorama, a generic failure detection framework that leverages and enhances system observability to detect complex production failures. It does so by breaking detection boundaries and systematically extracting critical observations from diverse components. Panorama provides unified abstractions and APIs to report observations, and a distributed service to selectively exchange observations. Also, importantly, Panorama keeps the burden on developers low by automatically inserting report-generation code based on offline static analysis. In this way, Panorama automatically converts every component into an observer of the components it interacts with. This construction of *in-situ* observers differentiates Panorama from traditional distributed crash failure detection services [34, 47], which only measure superficial failure indicators.

In applying Panorama to real-world system software, we find some common design patterns that, if not treated appropriately, can reduce observability and lead to misleading observations. For example, if a requester submits requests to a provider, but an indirection layer temporarily buffers the request, the request may appear successful even though the provider has failed. This can cause the requester to report positive evidence about the provider. We study such common design patterns and characterize their impact on system observability (§4). Based on this, we enhance Panorama to recognize these patterns and avoid their effects on observability.

For failure detection, Panorama includes a decision engine to reach a verdict on the status of each component based on reported observations. Because these reports come from errors and successes in the execution paths of requester components instead of artificial, non-service signals, our experience suggests that a simple decision algorithm suffices to reliably detect complex failures.

We have implemented the Panorama system in Go and the static analyzer on top of Soot [46] and AspectJ [2]. Our experiences show that Panorama is easy to integrate with popular distributed systems including ZooKeeper, Cassandra, HDFS, and HBase. Panorama significantly outperforms existing failure detectors in that: (1) it detects crash failures faster; (2) it detects 15 **real-world** gray failures in less than 7 s each, whereas other detectors only detect one in 86 s; (3) Panorama not only detects, but also *locates* failures. Our experiments also show that Panorama is resilient to transient failures and is stable in normal operations. Finally, Panorama introduces only minor overhead (less than 3%) to the systems we evaluate it on.

2 Problem Statement

We consider failure detection in the context of a large distributed system S composed of several subsystems. Each subsystem has multiple components. In total, S contains n processes $P_1, P_2, \ldots, P_n$, each with one or more threads. The whole system lies within a single administrative domain but the code for different system components may be developed by different teams. For example, a stor-

age system may consist of a front-end tier, a distributed lock service, a caching middleware, a messaging service, and a persistence layer. The latter subsystem include metadata servers, structured table servers, and extent data nodes. An extent data node may be multi-threaded, with threads such as a data receiver, a data block scanner, a block pool manager, and an IPC-socket watcher. We assume the components trust each other, collectively providing services to external untrusted applications.

The main goal of failure detection is to correctly report the status of each component; in this work the only components we consider are processes and threads. Traditional failure detectors focus on crash failure, i.e., using only statuses `UP` and `DOWN`. We aim to detect not only crash failure but also gray failure, in which components experience degraded modes "between" `UP` and `DOWN`. The quality of a failure detector is commonly characterized by two properties: *completeness*, which requires that if a component fails, a detector eventually suspects it; and *accuracy*, which requires that a component is not suspected by a detector before it fails. Quality is further characterized by *timeliness*, i.e., how fast true failures are detected. Failure detectors for production systems should also have good *localization*, i.e., ease of pinpointing each failure in a way that enables expedient corrective action.

3 Panorama System

3.1 Overview

At a high level, Panorama takes a collaborative approach: It gathers observations about each component from different sources in real time to detect complex production failures. Collaborative failure detection is not a new idea. Many existing crash-failure detectors such as membership services exchange detection results among multiple components using protocols like gossip [47]. But, the scope of where the detection is done is usually limited to component instances with similar functionality or roles in a particular layer. Panorama pushes the detection scope to an extreme by allowing any thread in any process to report evidence, regardless of its role, layer, or subsystem. The resulting diverse sources of evidence enhance the observability of complex failures.

More importantly, instead of writing separate monitoring code that measures superficial signals, Panorama's philosophy is to leverage *existing code* that lies near the boundaries between different components. Examples of such code include when one thread calls another, and when one process makes an RPC call to another. This captures first-hand observations, especially runtime errors that are generated from the executions of these code regions in production. When Panorama reports a failure, there is concrete evidence and context to help localize where the failure happened.

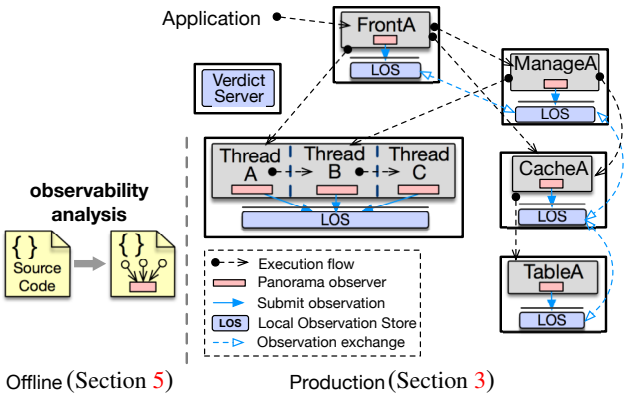


Figure 1: Overview of Panorama. Each Panorama instance runs at the same endpoint with the monitored component.

Figure 1 shows an overview of Panorama. Panorama is a generic detection service that can be plugged into any component in a distributed system. It provides unified abstractions to represent observations about a component's status, and a library for reporting and querying detection results. For scalability, we use a decentralized architecture: for each P_i in a monitored system, a co-located Panorama instance (a separate process) maintains a Local Observation Store (LOS) that stores all the observations that are made either by or about P_i. A local decision engine in the instance analyzes the observations in that LOS and makes a judgment about the process's status. A central verdict server allows easy querying of, and arbitration among, these decentralized LOSes.

The Panorama service depends on many *logical observers* within the running components in the monitored system. Unlike traditional failure detectors, these logical observers are *not* dedicated threads running detection checks. Rather, they are diverse hooks injected into the code. These hooks use a thin library to collect and submit observations to the LOS via local RPC calls. They are inserted offline by a tool that leverages static analysis (§5). To achieve timeliness, the observations are reported in real time as P_i executes. Panorama observers collect evidence not only about the locally attached component, but, more importantly, about other components that the observer interacts with. However, if P_i never interacts with P_j, P_i will not put observations about P_j into its LOS. Panorama runs a dissemination protocol to exchange observations among a clique of LOSes that share common interaction components.

3.2 Abstractions and APIs

To be usable by arbitrary distributed system components, Panorama must provide a unified way to encapsulate ob-

Component	a process or thread
Subject	a component to be monitored
Observer	a component monitoring a subject
Status	the health situation of a subject
Observation	evidence an observer finds of a subject's status
Context	what an observer was doing when it made an observation
Verdict	a decision about a subject's status, obtained by summarizing a set of observations of it

Table 1: Abstractions and terms used in Panorama.

servations for reporting. We now describe our core abstractions and terms, summarized in Table 1.

As discussed earlier, the only components we consider are processes and threads. A component is an *observer* if it makes observations and a *subject* if it is observed; a component may be both an observer and a subject. A *status* is a categorization of the health of a subject; it can be only a small pre-determined set of values, including `HEALTHY`, `DEAD`, and a few levels of `UNHEALTHY`. Another possible value is `PENDING`, the meaning and use of which we will discuss in §5.4.

When an observer sees evidence of a subject's status, that constitutes an *observation*. An observation contains a timestamp of when the observation occurred, the identities of the observer and subject, and the inferred status of the subject. It also contains a *context* describing what the observer was doing when it made the observation, at a sufficient granularity to allow Panorama to achieve fine-grained localization of failures. For instance, the context may include the method the observer was running, or the method's class; the API call the observer was making to the subject; and/or the type of operation, e.g., short-circuit read, snapshot, or row mutation. A *verdict* is a summary, based on a decision algorithm, of a set of observations of the same subject.

Each Panorama instance provides an API based on the above abstractions. It can be invoked by a local component, by another Panorama instance, or by an administration tool. When a component decides to use Panorama, it registers with the local Panorama instance and receives a handle to use for reporting. It reports observations using a local RPC `ReportObservation`; when it is done reporting it unregisters. A Panorama instance can register multiple local observers. If a component does not intend to report observations but merely wants to query component statuses, it need not register.

Each Panorama instance maintains a *watch list*: the set of subjects for which it keeps track of observations. By default, Panorama automatically updates this list to include the components that registered observers interact with. But, each observer can explicitly select subjects for this list using `StartObserving` and `StopObserving`. If another observer in another Panorama instance makes an observation about a subject in the watch list, that observation will be propagated to this instance with a remote RPC `LearnObservation`. Panorama calls `JudgeSubject` each time it collects a new observation, either locally or via remote exchange.

3.3 Local Observation Store

Each Panorama instance maintains a Local Observation Store (LOS) that stores all observation reports made by colocated components. The subjects of these reports include both local and remote components.

The LOS consists of two main structures: the raw observation store and the verdict table. The LOS partitions the raw observation store by subject into multiple tables for efficient concurrent access. Each record in a subject's table corresponds to a single observer; it stores a list of the n most recent observations of that subject made by that observer. The LOS is kept in memory to enable efficient access; asynchronously, its content is persisted to local database to preserve the full observation history, for facilitating troubleshooting later. The raw observation store is synchronized with that of other Panorama instances that share common subjects. Therefore, an LOS contains observations made both locally and remotely.

A local decision engine analyzes the raw observation store to reach a verdict for each subject. This decision result is stored in the verdict table, keyed by subject. The verdict table is *not* synchronized among Panorama instances because it does not have to be: the decision algorithm is deterministic. In other words, given synchronized raw observations, the verdict should be the same. To enable convenient queries over the distributed verdict tables to, e.g., arbitrate among inconsistent verdicts, Panorama uses a central verdict server. Note, though, that the central verdict server is not on any critical path.

Including old observations in decisions can cause misleading verdicts. So, each observation has a Time-to-Live parameter, and a background garbage collection (GC) task runs periodically to retire old observations. Whenever GC changes the observations of a subject, the decision engine re-computes the subject's verdict.

3.4 Observers

Panorama does not employ dedicated failure detectors. Instead, it leverages code logic in existing distributed-system components to turn them into in-situ *logical* observers. Each logical observer's main task is still to provide its original functionality. As it executes, if it encounters an error related to another component, in addition to handling the error it will also report it as an observation to Panorama. There are two approaches to turn

a component into a Panorama observer. One is to insert Panorama API hooks into the component's source code. Another is to integrate with the component's logs by continuously parsing and monitoring log entries related to other components. The latter approach is transparent to components but captures less accurate information. We initially adopted the latter approach by adding plug-in support in Panorama to manage log-parsing scripts. But, as we applied Panorama to more systems, maintaining these scripts became painful because their logging practices differed significantly. Much information is also unavailable in logs [50]. Thus, even though we still support logging integration, we mainly use the instrumentation approach. To relieve developers of the burden of inserting Panorama hooks, Panorama provides an offline analysis tool that does the source-code instrumentation automatically. §4 describes this offline analysis.

3.5 Observation Exchange

Observations submitted to the LOS by a local observer only reflect a partial view of the subject. To reduce bias in observations, Panorama runs a dissemination protocol to propagate observations to, and learn observations from, other LOSes. Consequently, for each monitored subject, the LOS stores observations from multiple observers. The observation exchange in Panorama is only among cliques of LOSes that share a subject. To achieve selective exchange, each LOS keeps a *watch list*, which initially contains only the local observer. When a local observer reports an observation to the LOS, the LOS will add the observation's subject to the watch list to indicate that it is now interested in others' observations about this subject. Each LOS also keeps an *ignore list* for each subject, which lists LOSes to which it should not propagate new observations about that subject. When a local observation for a new subject appears for the first time, the LOS does a one-time broadcast. LOSes that are not interested in the observation (based on their own watch lists) will instruct the broadcasting LOS to include them in its ignore list. If an LOS later becomes interested in this subject, the protocol ensures that the clique members remove this LOS from their ignore lists.

3.6 Judging Failure from Observations

With numerous observations collected about a subject, Panorama uses a decision engine to reach a verdict and stores the result in the LOS's verdict table. A simple decision policy is to use the latest observation as the verdict. But, this can be problematic since a subject experiencing intermittent errors may be treated as healthy. An alternative is to reach an unhealthy verdict if there is *any* recent negative observation. This could cause one biased observer, whose negative observation is due to its own issue, to mislead others.

We use a bounded-look-back majority algorithm, as follows. For a set of observations about a *subject*, we first group the observations by the unique *observer*, and analyze each group separately. The observations in a group are inspected from latest to earliest and aggregated based on their associated *contexts*. For an observation being inspected, if its *status* is different than the previously recorded status for that context, the look-back of observations for that context stops after a few steps to favor newer statuses. Afterwards, for each recorded context, if either the latest status is unhealthy or the healthy status does not have the strict majority, the verdict for that context is unhealthy with an aggregated severity level.

In this way, we obtain an analysis summary for each context in each group. To reach a final verdict for each context across all groups, the summaries from different observers are aggregated and decided based on a simple majority. Using group-based summaries allows incremental update of the verdict and avoids being biased by one observer or context in the aggregation. The decision engine could use more complex algorithms, but we find that our simple algorithm works well in practice. This is because most observations collected by Panorama constitute strong evidence rather than superficial signals.

The `PENDING` status (Section 4.3) needs additional handling: during the look-back for a context, if the current status is `HEALTHY` and the older status is `PENDING`, that older `PENDING` status will be skipped because it was only temporary. In other words, that partial observation is now complete. Afterwards, a `PENDING` status with occurrences exceeding a threshold is downgraded to `UNHEALTHY`.

4 Design Pattern and Observability

The effectiveness of Panorama depends on the hooks in observers. We initially designed a straightforward method to insert these hooks. In testing it on real-world distributed systems, however, we found that component interactions in practice can be complex. Certain interactions, if not treated appropriately, will cause the extracted observations to be misleading. In this section, we first show a gray failure that our original method failed to detect, and then investigate the reason behind the challenge.

4.1 A Failed Case

In one incident of a production ZooKeeper service, applications were experiencing many lock timeouts [23]. An engineer investigated the issue by checking metrics in the monitoring system and found that the number of connections per client had significantly increased. It ini-

tially looked like a resource leak in the client library, but the root cause turned out to be complicated.

The production environment used IPSec to secure inter-host traffic, and a Linux kernel module used Intel AES instructions to provide AES encryption for IPSec. But this kernel module could occasionally introduce data corruption with Xen paravirtualization, for reasons still not known today. Typically the kernel validated packet checksums and dropped corrupt packets. But, in IPSec, two checksums exist: one for the IP payload, the other for the encrypted TCP payload. For IPSec NAT-T mode, the Linux kernel did not validate the TCP payload checksum, thereby permitting corrupt packets. These were delivered to the ZooKeeper leader, including a corrupted length field for a string. When ZooKeeper used the length to allocate memory to deserialize the string, it raised an out-of-memory (OOM) exception.

Surprisingly, when this OOM exception happened, ZooKeeper continued to run. Heartbeats were normal and no leader re-election was triggered. When evaluating this incident in Panorama, no failure was reported either. We studied the ZooKeeper source code to understand why this happened. In ZooKeeper, a request is first picked up by the listener thread, which then calls the `ZooKeeperServer` thread that further invokes a chain of `XXXRequestProcessor` threads to process the request. The OOM exception happens in the `PrepRequestProcessor` thread, the first request processor. The `ZooKeeperServer` thread invokes the interface of the `PrepRequestProcessor` as follows:

```
1  try {
2    firstProcessor.processRequest(si);
3  } catch (RequestProcessorException e) {
4    LOG.error("Unable to process request: " + e);
5  }
```

If the execution passes line 2, it provides positive evidence that the `PrepRequestProcessor` thread is healthy. If, instead, the execution reaches line 4, it represents negative evidence about `PrepRequestProcessor`. But with the Panorama hooks inserted at both places, no negative observations are reported. This is because the implementation of the `processRequest` API involves an indirection: it simply puts a request in a queue and immediately returns. Asynchronously, the thread polls and processes the queue. Because of this design, even though the OOM exception causes the `PrepRequestProcessor` thread to exit its main loop, the `ZooKeeperServer` thread is still able to call `processRequest` and is unable to tell that `PrepRequestProcessor` has an issue. The hooks are only observing the status of the indirection layer, i.e., the queue, rather than the `PrepRequestProcessor` thread. Thus, negative observations only appear when the request queue cannot insert new items; but, by default, its capacity is `Integer.MAX_VALUE`!

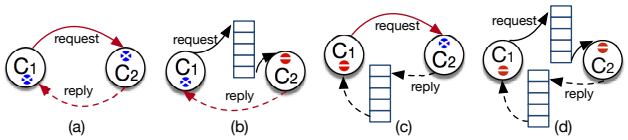


Figure 2: Design patterns of component interactions and their impact on failure observability. ⛒ means that failure is observable to the other component, and ⛔ means that failure is unobservable to it.

4.2 Observability Patterns

Although the above case is a unique incident, we extrapolate a deeper implication for failure detection: certain design patterns can undermine failure observability in a system and thereby pose challenges for failure detection. To reveal this connection, consider two components C_1 and C_2 where C_1 makes requests of C_2. We expect that, through this interaction, C_1 and C_2 should be able to make observations about each other's status. However, their style of interaction can have a significant effect on this observability.

We have identified the following four basic patterns of interaction (Figure 2), each having a different effect on this observability. Interestingly, we find examples of all four patterns in real-world system software.

(a) No indirection. Pattern (a) is the most straightforward. C_1 makes a request to C_2, then C_2 optionally replies to C_1. This pattern has the best degree of observability: C_1 can observe C_2 from errors in its request path; C_2 can also observe C_1 to some extent in its reply path. Listing 1 shows an example of this pattern. In this case, C_1 is the follower and C_2 is the leader. C_1 first contacts C_2, then C_2 sends C_1 a snapshot or other information through an input stream. Failures are observed via errors or timeouts in the connection, I/O through the input stream, and/or reply contents.

(b) Request indirection. A level of indirection exists in the request path: when C_1 makes a request to C_2, an intermediate layer (e.g., a proxy or a queue) takes the request and replies to C_1. C_2 will later take the request from the intermediate layer, process it, and optionally reply to C_1 directly. This design pattern has a performance benefit for both C_1 and C_2. It also provides decoupling between their two threads. But, because of the indirection, C_1 no longer directly interacts with C_2 so C_2's observability is reduced. The immediate observation C_1 makes when requesting from C_2 does not reveal whether C_2 is having problems, since usually the request path succeeds as in the case in §4.1.

(c) Reply indirection. Pattern (c) is not intuitive. C_1 makes a request, which is directly handled by C_2, but the reply goes through a layer of indirection (e.g., a queue or a proxy). Thus, C_1 can observe issues in C_2 but C_1's ob-

servability to C_2 is reduced. One scenario leading to this pattern is when a component makes requests to multiple components and needs to collect more than one of their replies to proceed. In this case, replies are queued so that they can be processed en masse when a sufficient number are available. For example, in Cassandra, when a process sends digest requests to multiple replicas, it must wait for responses from R replicas. So, whenever it gets a reply from a replica, it queues the reply for later processing.

(d) Full indirection. In pattern (d), neither component directly interacts with the other so they get the least observability. This pattern has a performance benefit since all operations are asynchronous. But, the code logic can be complex. ZooKeeper contains an example: When a follower forwards a request to a leader, the request is processed asynchronously, and when the leader later notifies the follower to commit the request, that notification gets queued.

4.3 Implications

Pattern (a) has the best failure observability and is easiest for Panorama to leverage. The other three patterns are more challenging; placing observation hooks without considering the effects of indirection can cause incompleteness (though not inaccuracy) in failure detection (§2). That is, a positive observation will not necessarily mean the monitored component is healthy but a negative observation means the component is unhealthy. Pragmatically, this would be an acceptable limitation if the three indirection patterns were uncommon. However, we checked the cross-thread interaction code in several distributed systems and found, empirically, that patterns (a) and (b) are both pervasive. We also found that different software has different preferences, e.g., ZooKeeper uses pattern (a) frequently, but Cassandra uses pattern (b) more often.

This suggests Panorama should accommodate indirection in extracting observations. One solution is to instrument hooks in the indirection layer. But, we find that indirection layers in practice are implemented with various data structures and are often used for multiple purposes, making tracking difficult. We use a simple but robust solution and describe it in §5.4.

5 Observability Analysis

To systematically identify and extract useful observations from a component, Panorama provides an offline tool that statically analyzes a program's source code, finds critical points, and injects hooks for reporting observations.

5.1 Locate Observation Boundary

Runtime errors are useful evidence of failure. Even if an error is tolerated by a requester, it may still indicate a critical issue in the provider. But, not all errors should be reported. Panorama only extracts errors generated when crossing component boundaries, because these constitute observations from the requester side. We call such domain-crossing function invocations *observation boundaries*.

The first step of observability analysis is to locate observation boundaries. There are two types of such boundaries: inter-process and inter-thread. An inter-process boundary typically manifests as a library API invocation, a socket I/O call, or a remote procedure call (RPC). Sometimes, it involves calling into custom code that encapsulates one of those three to provide a higher-level messaging service. In any case, with some domain knowledge about the communication mechanisms used, the analyzer can locate inter-process observation boundaries in source code. An inter-thread boundary is a call crossing two threads within a process. The analyzer identifies such boundaries by finding custom public methods in classes that extend the thread class.

5.2 Identify Observer and Observed

At each observation boundary, we must identify the observer and subject. Both identities are specific to the distributed system being monitored. For thread-level observation boundaries, the thread identities are statically analyzable, e.g., the name of the thread or class that provides the public interfaces. For process-level boundaries, the observer identity is the process's own identity in the distributed system, which is known when the process starts; it only requires one-time registration with Panorama. We can also usually identify the subject identity, if the remote invocations use well-known methods, via either an argument of the function invocation or a field in the class. A challenge is that sometimes, due to nested polymorphism, the subject identity may be located deep down in the type hierarchy. For example, it is not easy to determine if `OutputStream.write()` performs network I/O or local disk I/O. We address this challenge by changing the constructors of remote types (e.g., socket get I/O stream) to return a compatible wrapper that extends the return type with a subject field and can be differentiated from other types at runtime by checking if that field is set.

5.3 Extract Observation

Once we have observation boundaries, the next step is to search near them for *observation points*: program points that can supply critical evidence about observed components. A typical example of such an observation point is

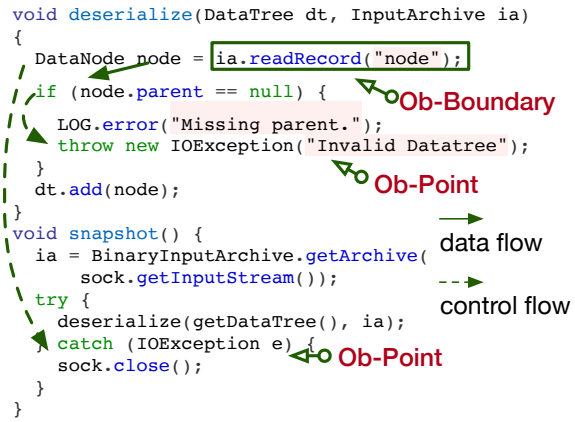


Figure 3: Observation points in direct interaction (§4.2).

an exception handler invoked when an exception occurs at an observation boundary.

To locate observation points that are exception handlers, a straightforward approach is to first identify the type of exceptions an observation boundary can generate, then locate the `catch` clauses for these types in code regions after the boundary. There are two challenges with this approach. First, as shown in Figure 3, an exception could be caught at the caller or caller's caller. Recursively walking up the call chain to locate the clause is cumbersome and could be inaccurate. Second, the type of exception thrown by the boundary could be a generic exception such as `IOException` that could be generated by other non-boundary code in the same `try` clause. These two challenges can be addressed by inserting a `try` just before the boundary and a `catch` right after it. This works but, if the observation boundaries are frequent, the excessive wrapping can cause non-trivial overhead.

The ideal place to instrument is the shared exception handler for adjacent invocations. Our solution is to add a special field in the base `Throwable` class to indicate the subject identity and the context, and to ensure boundary-generated exceptions set this field. Then, when an exception handler is triggered at runtime, we can check if this field is set, and if so treat it as an observation point. We achieve the field setting by wrapping the outermost function body of each boundary method with a `try` and `catch`, and by rethrowing the exception after the hook. Note that this preserves the original program semantics.

Another type of observation point we look for is one where the program handles a response received from across a boundary. For example, the program may raise an exception for a missing field or wrong signature in the returned `DataNode` in Figure 3, indicating potential partial failure or corrupt state in the remote process. To locate these observation points, our analyzer performs intra-procedural analysis to follow the data flow of responses from a boundary. If an exception thrown is control-dependent on the response, we consider it an observation point, and we insert code to set the subject/context field before throwing the exception just as described earlier. This data-flow analysis is conservative: e.g., the code `if (a + b > 100) {throw Exception("unexpected");}`, where `a` comes from a boundary but `b` does not, is not considered an observation point because the exception could be due to `b`. In other words, our analysis may miss some observation points but will not locate wrong observation points.

So far, we have described negative observation points, but we also need mechanisms to make positive observations. Ideally, each successful interaction across a boundary is an observation point that can report positive evidence. But, if these boundaries appear frequently, the positive observation points can be excessive. So, we coalesce similar positive observation points that are located close together.

For each observation point, the analyzer inserts hooks to discover evidence and report it. At each negative observation point, we get the subject identity and context from the modified exception instance. We statically choose the status; if the status is to be some level of `UNHEALTHY` then we set this level based on the severity of the exception handling. For example, if the exception handler calls `System.exit()`, we set the status to a high level of `UNHEALTHY`. At each positive observation point, we get the context from the nearby boundary and also statically choose the status. We immediately report each observation to the Panorama library, but the library will typically not report it synchronously. The library will buffer excessive observations and send them in one aggregate message later.

5.4 Handling Indirection

As we discussed in §4, observability can be reduced when indirection exists at an observation boundary. For instance, extracted observations may report the subject as healthy while it is in fact unhealthy. The core issue is that indirection *splits* a single interaction between components among multiple observation boundaries. A successful result at the first observation boundary may only indicate partial success of the overall interaction; the interaction may only truly complete later, when, e.g., a callback is invoked, or a condition variable unblocks, or a timeout occurs. We must ideally wait for an interaction to complete before making an observation.

We call the two locations of a split interaction the *ob-origin* and *ob-sink*, reflecting the order they're encountered. Observations at the ob-origin represent positive but temporary and weak evidence. For example, in Figure 4, the return from `sendRR` is an ob-origin. Where the

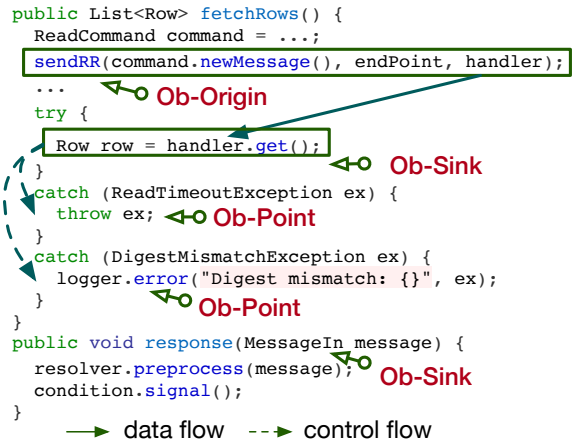


Figure 4: Observation points when indirection exists (§4.2).

callback of `handler`, `response`, is invoked, it is an ob-sink. In addition, when the program later blocks waiting for the callback, e.g., `handler.get`, the successful return is also an ob-sink. If an ob-origin is properly matched with an ob-sink, the positive observation becomes complete and strong. Otherwise, an outstanding ob-origin is only a weak observation and may degrade to a negative observation, e.g., when `handler.get` times out.

Tracking an interaction split across multiple program locations is challenging given the variety of indirection implementations. To properly place hooks when indirection exists, the Panorama analyzer needs to know what methods are asynchronous and the mechanisms for notification. For instance, a commonly used one is Java FutureTask [40]. For custom methods, this knowledge comes from specifications of the boundary-crossing interfaces, which only requires moderate annotation. With this knowledge, the analyzer considers an ob-origin to be immediately after any call site of an asynchronous interface. We next discuss how to locate ob-sinks.

We surveyed the source code of popular distributed systems and found the majority of ob-sinks fall into four patterns: (1) invoking a callback-setting method; (2) performing a blocking wait on a callback method; (3) checking a completion flag; and (4) reaching another observation boundary with a third component, in cases when a request must be passed on further. For the first two patterns, the analyzer considers the ob-sink to be before and after the method invocation, respectively. For the third pattern, the analyzer locates the spin-loop body and considers the ob-sink to be immediately after the loop. The last pattern resembles SEDA [48]: after A asynchronously sends a request to B, B does not notify A of the status after it finishes but rather passes on the request to C. Therefore, for that observation boundary in B, the analyzer needs to not only insert a hook for C but also treat it as an ob-sink for the A-to-B interaction.

When our analyzer finds an ob-origin, it inserts a hook that submits an observation with the special status `PENDING`. This means that the observer currently only sees weak positive evidence about the subject's status, but expects to receive stronger evidence shortly. At any ob-sink indicating positive evidence, our analyzer inserts a hook to report a `HEALTHY` observation. At any ob-sink indicating negative evidence, the analyzer inserts a hook to report a negative observation.

To link an ob-sink observation with its corresponding ob-origin observation, these observations must share the same subject and context. To ensure this, the analyzer uses a similar technique as in exception tracking. It adds a special field containing the subject identity and context to the callback handler, and inserts code to set this field at the ob-origin. If the callback is not instrumentable, e.g., because it is an integer resource handle, then the analyzer inserts a call to the Panorama library to associate the handle with an identity and context.

Sometimes, the analyzer finds an ob-origin but cannot find the corresponding ob-sink or cannot extract the subject identity or context. This can happen due to either lack of knowledge or the developers having forgotten to check for completion in the code. In such a case, the analyzer will not instrument the ob-origin, to avoid making misleading `PENDING` observations.

We find that ob-origin and ob-sink separation is useful in detecting not only issues involving indirection but also liveness issues. To see why, consider what happens when A invokes a boundary-crossing blocking function of B, and B gets stuck so the function never returns. When this happens, even though A witnesses B's problem, it does not get a chance to report the issue because it never reaches the observation point following the blocking call. Inserting an ob-origin before the function call provides evidence of the liveness issue: LOSes will see an old `PENDING` observation with no subsequent corresponding ob-sink observation. Thus, besides asynchronous interfaces, call sites of synchronous interfaces that may block for long should also be included in the ob-origin set.

6 Implementation

We implemented the Panorama service in $\sim 6,000$ lines of Go code, and implemented the observability analyzer (§5) using the Soot analysis framework [46] and the AspectJ instrumentation framework [2].

We defined Panorama's interfaces using protocol buffers [7]. We then used the gRPC framework [5] to build the RPC service and to generate clients in different languages. So, the system can be easily used by various components written in different languages. Panorama provides a thin library that wraps the gRPC client for

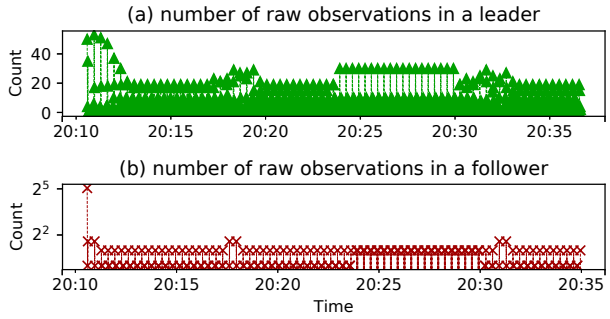


Figure 5: Number of raw observations in two Panorama observers. Each data point represents one second.

efficient observation reporting; each process participating in observation reporting is linked with this library. The thin library provides features such as asynchronous reporting, buffering and aggregation of frequent observations, identity resolution, rate limiting, quick cancellation of `PENDING` statuses, and mapping of ob-sink handles (§5.4). So, most operations related to observation reporting do not directly trigger local RPC calls to Panorama; this keeps performance impact low.

7 Evaluation

In this section, we evaluate our Panorama prototype to answer several key questions: (1) Can observations be systematically captured? (2) Can observation capturing detect regular failures? (3) Can Panorama detect production gray failures? (4) How do transient failures affect Panorama? (5) How much overhead does an observer incur by participating in the Panorama service?

7.1 Experiment Setup

We run our experiments in a cluster of 20 physical nodes. Each machine has a 2.4 GHz 10-core Intel Xeon E5-2640v4 CPU, 64 GB of RAM, and a 480 GB SATA SSD; they all connect to a single 10 Gbps Ethernet switch. They run Ubuntu 16.04 with Linux kernel version 4.4.0. We evaluate Panorama with four widely-used distributed systems: ZooKeeper, Hadoop, HBase, and Cassandra. HBase uses HDFS for storing data and ZooKeeper for coordination, so an HBase setup resembles a service with multiple subsystems. We continuously exercise these services with various benchmark workloads to represent an active production environment.

7.2 Integration with Several Systems

Panorama provides a generic observation and failure detection service. To evaluate its generality, we apply it to ZooKeeper, HDFS, Hadoop, HBase, and Cassandra, at both process and thread level. The integration is successful without significant effort or changes to the system design. Our simple abstractions and APIs (§3.2) naturally support various types of failure evidence in each system. For instance, we support semantic errors, such as responses with missing signatures; generic errors, such as remote I/O exceptions; and liveness issues, such as indefinite blocking or custom time-outs. The integration is enabled by the observability analyzer (§5). In applying the analyzer to a system, we need annotations about what boundary-crossing methods to start with, what methods involve indirection, and what patterns it uses (§5.4). The annotation effort to support this is moderate (Table 2). HDFS requires the most annotation effort, which took one author about 1.5 days to understand the HDFS source code, identify the interfaces and write annotation specification. Fortunately, most of these boundary-crossing methods remain stable over releases. When running the observability analysis, Cassandra is more challenging to analyze compared to the others since it frequently uses indirection. On the other hand, its mechanisms are also well-organized, which makes the analysis systematic. The observability analysis is mainly intra-procedural and can finish instrumentation within 10 seconds for each of the four systems (Table 2). Figure 5 shows the observations collected from two instrumented processes in ZooKeeper. The figure also shows that the observations made change as the observer executes, and depend on the process's interaction patterns.

	ZooKeeper	Cassandra	HDFS	HBase
Annotations	24	34	65	16
Analysis Time	4.2	6.8	9.9	7.5

Table 2: Annotations and analysis time (in seconds).

7.3 Detection of Crash Failures

Panorama aims to detect complex failures not limited to fail-stop. As a sanity check on the effectiveness of its detection capability, we first evaluate how well Panorama detects fail-stop failures. To measure this, we inject various fail-stop faults including process crashes, node shutdowns, and network disconnections. Table 3 shows the detection time for ten representative crash-failure cases: failures injected into the ZooKeeper leader, ZooKeeper follower, Cassandra data node, Cassandra seed node, HDFS name node, HDFS data node, HBase master and HBase regionserver. We see that with Panorama the observers take less than 10 s to detect all ten cases, and indeed take less than 10 ms to detect all ZooKeeper failures. The observers make the observations leading to these detections when, while interacting with the

Detector	Crash Failure Injection Site							
	ZooKeeper		Cassandra		HDFS		HBase	
	leader	follower	seed	datanode	namenode	datanode	master	regionserver
Built-in	13 ms	3 ms	28 s	26 s	708 ms	30 s (12 min*)	11 s	102 ms
Panorama	8 ms	2 ms	8 s	9 s	723 ms	6 s	1.5 s	102 ms

Table 3: Crash-failure detection time. **The name node marks the data node stale in 30 s, and dead in 12 min.*

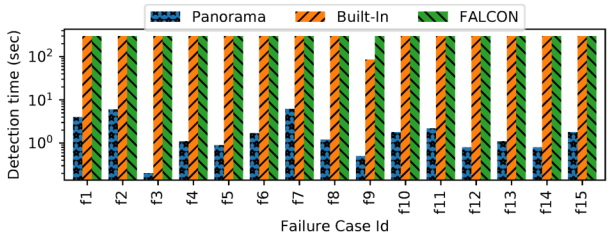


Figure 6: Detection time for gray failures in Table 4.

ID	System	Fault Synopsis
f1	ZooKeeper	faulty disk in leader causes cluster lock-up
f2	ZooKeeper	transient network partition leads to prolonged failures in serving requests
f3	ZooKeeper	corrupted packet in de-serialization
f4	ZooKeeper	transaction thread exception
f5	ZooKeeper	leader fails to write transaction log
f6	Cassandra	response drop blocks repair operations
f7	Cassandra	stale data in leads to wrong node states
f8	Cassandra	streaming silently fail on unexpected error
f9	Cassandra	commitlog executor exit causes GC storm
f10	HDFS	thread pool exhaustion in master
f11	HDFS	failed pipeline creation prevents recovery
f12	HDFS	short circuit reads blocked due to death of domain socket watcher
f13	HDFS	blockpool fails to initialize but continues
f14	HBase	dead root drive on region server
f15	HBase	replication stalls with empty WAL files

Table 4: Evaluated *real-world* gray failures. In all cases, some severe service disruption occured (e.g., all create requests failed) while the failing component was perceived to be healthy.

failed components, they experience either request/response time-outs or I/O exceptions.

As a basis for comparison, we also measure failure detection time when using the failure detectors built into these systems. We find that for ZooKeeper, Panorama detects the failures slightly faster than the built-in detector, while for Cassandra, HDFS datanode and HBase master, Panorama achieves much faster detection time. This is because, to tolerate asynchrony, Cassandra and HDFS use conservative settings for declaring failures based on loss of heartbeats. For HDFS namenode, we use a High-Availability setup that leverages ZooKeeper for failure detection (when a ZooKeeper ephemeral node expires). Under this setup, the built-in detector achieves a slightly faster time than Panorama because the ZooKeeper service is co-located with HDFS, whereas Panorama's detection is from observations made by remote datanodes.

7.4 Detection of Gray Failures

To evaluate Panorama's ability to detect complex failures, we reproduce 15 **real-world** production gray failures from ZooKeeper, HDFS, HBase, and Cassandra, described in Table 4. Each of these caused severe service disruption, e.g., all write requests would fail. Worse still, in each case the system was perceived as healthy, so no recovery actions were taken during the resulting outage.

Panorama is able to detect the gray failure for **all** 15 cases. Figure 6 shows Panorama's detection time (in seconds) for each case. We often find that a failure is observed and reported by multiple observers; we use the first failure observation's timestamp in a final verdict as the detection time. The detection times have a minimum of 0.2 s and a maximum of 7 s, with the majority smaller than 3 s. The intra-process observers tend to capture failure evidence faster than the inter-process observers. For all cases, the failure evidence clearly stands out in the observations collected about the sick process, so the decision algorithm (§3.6) requires no special tuning.

We compare Panorama with three baselines: the system's built-in failure detector, Falcon [34], and the φ accrual detector [29]. As shown in Figure 6, in all but one case, no baseline detects the gray failure within 300 s. That one case is f9, where Cassandra's built-in detector, a form of the φ detector with some application state, reports failure after 86 s when the partial fault of the Cassandra commitlog executor component eventually degrades into a complete failure due to uncommitted writes piling up on the JVM heap and causing the process to spend most of its time doing garbage collection.

Figure 7 shows a detailed timeline of the detection of gray failure f1. We see that the observers (in this case the followers) quickly gather failure evidence while interacting with the unhealthy leader. Also, when the leader's fault is gone, those observers quickly gather positive evi-

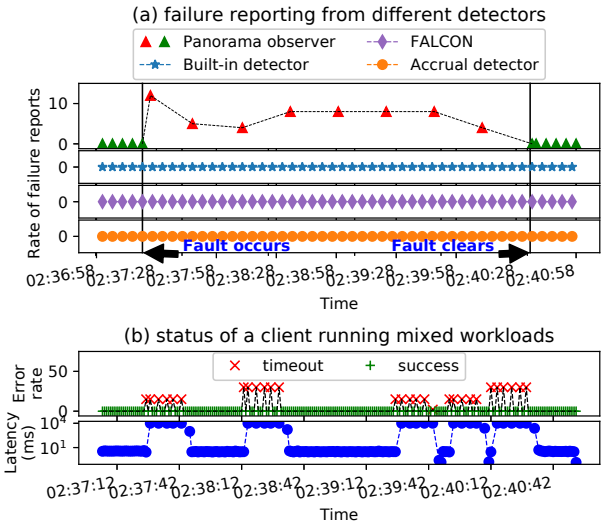


Figure 7: Timeline in detecting gray failure f1 from Table 4.



Figure 8: Verdict during transient failures.

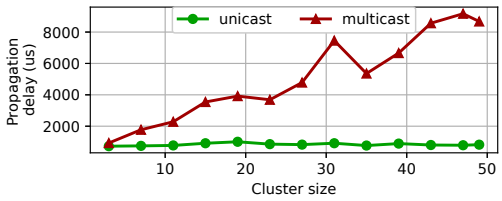


Figure 9: Scalability of observation propagation latency. "unicast": propagate an observation to a single Panorama instance; "multicast": propagate an observation to all interested Panorama instances.

dence that clears the failure observation. During the failure period, no other baseline reports failure. Figure 7 also shows the view from a ZooKeeper client that we run continuously throughout the experiment as a reference. We can see Panorama's reporting closely matches the experience of this client. Interestingly, since the gray failure mainly impacts write requests but the client executes a mixture of read and write requests, its view is not very stable; nevertheless, Panorama consistently reports a verdict of `UNHEALTHY` during the failure period.

7.5 Fault Localization

In addition to detecting the 15 production failures quickly, Panorama also pinpoints each failure with detailed context and observer (§3.2) information. This localization capability allows administrators to interpret the detection results with confidence and take concrete actions. For example, in detecting the crash failure in the ZooKeeper follower, the verdict for the leader is based on observations such as `|peer@3,peer@5,peer@8| 2018-03-23T02:28:58.873 {Learner: U,RecvWorker: U,QuorumCnxManager: U}`, which identify the observer as well as the contexts `Learner`, `RecvWorker`, and `QuorumCnxManager`. In detecting gray failure f1, the negative observations of the unhealthy leader are associated with three contexts `SerializeUtils`, `DataTree`, and `StatPersisted`; this localizes the failure to the serialization thread in leader.

7.6 Transient Failure, Normal Operations

Because Panorama can gather observations from any component in a system, there is a potential concern that noisy observations will lead to many false alarms. But, empirically, we find that this does not happen. The Panorama analyzer assigns the context of an observation properly to avoid falsely aggregating observations made in interacting with different functionalities of a complex process. The simple decision algorithm in §3.6 is robust enough to prevent a few biased observers or transient failures from dominating the verdict. Figure 8 shows the verdict for the ZooKeeper leader in an experiment. A few followers report transient faults about the leader in one context, so Panorama decides on a negative verdict. But, within a few seconds, the verdict changes due to positive observations and expiration of negative observations. Panorama then judges the leader as healthy for the remainder of the experiment, which matches the truth.

We deploy Panorama with ZooKeeper and run for 25 hours, during which multiple ZooKeeper clients continuously run various workloads non-stop to emulate normal operations in a production environment. In total, Panorama generates 797,219 verdicts, with all but 705 (0.08%) of them being `HEALTHY`; this is a low false alarm rate. In fact, all of the negative observations are made in the first 22 seconds, during which the system is bootstrapping and unstable. After the 22 seconds, no negative observations are reported for the remaining 25 hours.

We also inject minor faults including overloaded component, load spike and transient network partition that are modeled after two production ZooKeeper and HDFS traces. These minor faults do not affect the regular service. We find Panorama overall is resilient to these noises in reaching a verdict. For example, an overloaded ZooKeeper follower made a series of misleading obser-

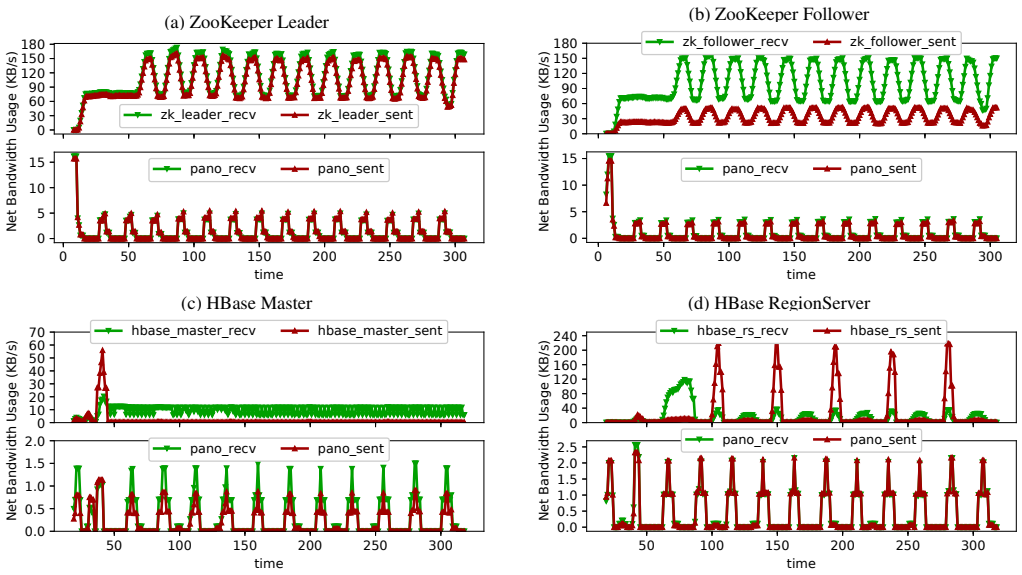


Figure 10: Network bandwidth usage of the Panorama instance and its monitored component.

Report	ReportAsync	Judge	Propagate
114.6 µs	0.36 µs	109.0 µs	776.3 µs

Table 5: Average latency of major operations in Panorama.

vations that the leader is `UNHEALTHY`. But these biased observations from a single observer did not result in a verdict of `UNHEALTHY` status for the leader. When there were many such overloaded followers, however, the leader was falsely convicted as `UNHEALTHY` even though the actual issues were within the observers.

7.7 Performance

Table 5 shows microbenchmark results: how long four major operations in Panorama take on average. Reporting an observation to Panorama only requires a local RPC, so the average latency of reporting is fast (around 100 µs). And, the asynchronous API for reporting takes even less time: on average less than 1 µs. Propagation of an observation to another Panorama instance takes around 800 µs. Figure 9 shows how the propagation latency changes as the cluster size increases.

When a Panorama instance is active, the CPU utilization attributable to it is on average 0.7%. For each monitored subject, the number of observations kept in LOS is bounded so the memory usage is close to a constant. Thus, the total memory usage depends on the number of monitored subjects. When we measure the ZooKeeper deployment with Panorama, and find that the heap memory allocation stabilizes at ∼7 MB for a moderately active instance, and at ∼46 MB for a highly active instance. The network bandwidth usage of Panorama instance for exchanging observations is small compared to the bandwidth usage of the monitored components (Figure 10).

System	Latency		Throughput	
	Read	Write	Read	Write
ZK	69.5 µs	1435 µs	14 402 op/s	697 op/s
ZK+	70.6 µs	1475 µs	14 181 op/s	678 op/s
C*	677 µs	680 µs	812 op/s	810 op/s
C*+	695 µs	689 µs	802 op/s	804 op/s
HDFS	51.0 s	61.0 s	423 MB/s	88 MB/s
HDFS+	52.5 s	62.2 s	415 MB/s	86 MB/s
HBase	746 µs	1682 µs	1172 op/s	549 op/s
HBase+	748 µs	1699 µs	1167 op/s	542 op/s

Table 6: Performance of the original system versus the performance of the system instrumented with Panorama hooks (System+). ZK stands for ZooKeeper and C* stands for Cassandra. The latency results for HDFS are total execution times.

We test the end-to-end request latency and throughput impact of integrating with Panorama for HDFS, ZooKeeper, HBase, and Cassandra, using YCSB [16], DFSIO and a custom benchmark tool. Table 6 shows the results. The latency increase and throughout decrease for each system is below 3%. We achieve this low overhead because the reporting API is fast and because most hooks are in error-handling code, which is not triggered in normal operation. The positive-observation hooks lie in the normal execution path, but their cost is reduced by coalescing the hooks with the analyzer (§5.3) and batching the reporting with the thin client library. Without this optimization, the performance overhead can be up to 18%.

8 Discussion and Limitations

Panorama proposes a new way of building failure detection service by constructing *in-situ* observers. The evaluation results demonstrate the effectiveness of leveraging observability for detecting complex production failures. The process of integrating Panorama with real-world distributed systems also makes us realize how the diverse programming paradigms affect systems observability. For example, HDFS has a method `createBlockOutputStream` that takes a list of data nodes as argument and creates a pipeline among them; if this method fails, it indicates one of the data nodes in the pipeline is problematic. From observability point of view, if a negative evidence is observed through this method, it is associated with multiple possible subjects. Fortunately, an `errorIndex` variable is maintained internally to indicate which data node causes the error, which can be used to determine the exact subject. It is valuable to investigate how to modularize a system and design its interfaces to make it easier to capture failure observability.

There are several limitations of Panorama that we plan to address in future work. First, Panorama currently focuses on failure detection. To improve end-to-end availability, we plan to integrate the detection results with failure recovery actions. Second, Panorama currently does not track causality. Enhancing observations with causality information will be useful for correctly detecting and pinpointing failing components in large-scale cascading failures. Third, we plan to add support for languages other than Java to the Panorama analyzer, and evaluate it with a broader set of distributed systems.

9 Related Work

Failure Detection. There is an extensive body of work on studying and improving failure detection for distributed systems [8, 13, 14, 20, 29, 47]. A recent prominent work in this space is Falcon [34], in which the authors argue that a perfect failure detector (PFD) can be built [9] by replacing end-to-end timeouts with layers of spies that can kill slow processes. Panorama is complimentary to these efforts, which mainly focus on detecting crash failures. Panorama's goal is to detect complex production failures [11, 25, 30]. In terms of approach, Panorama is unique in enhancing system observability by constructing *in-situ* observers in place of any component's code, instead of using dedicated detectors such as spies or sensors that are outside components' normal execution paths.

Monitoring and Tracing. Improving monitoring and tracing of production systems is also an active research area. Examples include Magpie [12], X-Trace [21], Dapper [45] and Pivot Tracing [35]. The pervasive metrics collected by these systems enhance system observability, and their powerful tracing capabilities may help Panorama better deal with the indirection challenge (§4). But they are massive and difficult to reason about [15, 37, 44]. Panorama, in contrast, leverages errors and exceptions generated from an observer's normal execution to report complex but serious failures.

Accountability. Accountability is useful for detecting Byzantine component behavior in a distributed system [28, 51]. PeerReview [27] provides accountability by having other nodes collecting evidence about the correctness of a node through their message exchanges. Panorama's approach is inspired by PeerReview in that it also leverages evidence about other components in a system. But Panorama mainly targets production gray failures instead of Byzantine faults. Unlike PeerReview, Panorama places observability hooks in the existing code of a component and does not require a reference implementation or a special protocol.

10 Conclusion

We present Panorama, a system for detecting production failures in distributed systems. The key insight enabling Panorama is that system observability can be enhanced by automatically turning each component into an observer of the other components with which it interacts. By leveraging these first-hand observations, a simple detection algorithm can achieve high detection accuracy. In building Panorama, we further discover observability patterns and address the challenge of reduced observability due to indirection. We implement Panorama and evaluate it, showing that it introduces minimal overhead to existing systems. Panorama can detect and localize 15 real-world gray failures in less than 7 s, whereas existing detectors only detect one of them in under 300 s. The source code of Panorama system is available at `https://github.com/ryanphuang/panorama`.

Acknowledgments

We thank the OSDI reviewers and our shepherd, Ding Yuan, for their valuable comments that improved the paper. We appreciate the support from CloudLab [43] for providing a great research experiment platform. We also thank Yezhuo Zhu for sharing ZooKeeper production traces and Jinfeng Yang for sharing HDFS production traces. This work was supported in part by a Microsoft Azure Research Award.

References

[1] Asana service outage on September 8th, 2016. https://blog.asana.com/2016/09/yesterdays-outage/.

[2] AspectJ, aspect-oriented extension to the Java programming language. https://www.eclipse.org/aspectj.

[3] GoCardless service outage on October 10th, 2017. https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october.

[4] Google Compute Engine incident 16007. https://status.cloud.google.com/incident/compute/16007.

[5] gRPC, a high performance, open-source universal RPC framework. https://grpc.io.

[6] Microsoft Azure status history. https://azure.microsoft.com/en-us/status/history.

[7] Protocol buffers. https://developers.google.com/protocol-buffers/.

[8] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, Apr. 2000.

[9] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, Monte Verità, Switzerland, May 2009. USENIX Association.

[10] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[11] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01. IEEE Computer Society, 2001.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI '04, San Francisco, CA, 2004. USENIX Association.

[13] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[14] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computing*, 51(5):561–580, May 2002.

[15] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 217–231, Broomfield, CO, 2014. USENIX Association.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, USA, 2010. ACM.

[17] J. Dean. Designs, lessons and advice from building large distributed systems, 2009. Keynote at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS).

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[19] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, 2013. ACM.

[20] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computing*, 52(2):99–112, Feb. 2003.

[21] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07, Cambridge, MA, 2007. USENIX Association.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, Bolton Landing, NY, USA, 2003. ACM.

[23] E. Gilman. PagerDuty production ZooKeeper service incident in 2014. https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet/.

[24] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210. ACM, 1989.

[25] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, USA, 2018. USENIX Association.

[26] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, SIGCOMM '15, pages 139–152, London, United Kingdom, 2015. ACM.

[27] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007. ACM.

[28] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 99–114, Nîmes, France, 2009. Springer-Verlag.

[29] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78. IEEE Computer Society, 2004.

[30] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, 2017. ACM.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '10, Boston, MA, 2010. USENIX Association.

[32] R. E. Kalman. On the general theory of control systems. *IRE Transactions on Automatic Control*, 4(3):110–110, December 1959.

[33] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, 2013. USENIX Association.

[34] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the Twenty-third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, 2011. ACM.

[35] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, Monterey, California, 2015. ACM.

[36] Microsoft. Office 365 service incident on November 13th, 2013. https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/.

[37] J. C. Mogul, R. Isaacs, and B. Welch. Thinking about availability in large service infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 12–17, Whistler, BC, Canada, 2017. ACM.

[38] D. Nadolny. Network issues can cause cluster to hang due to near-deadlock. https://issues.apache.org/jira/browse/ZOOKEEPER-2201.

[39] D. Nadolny. Debugging distributed systems. In *SREcon 2016*, Santa Clara, CA, Apr. 2016.

[40] Oracle. Java Future and FutureTask. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html.

[41] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[42] J. Postel. DoD Standard Transmission Control Protocol, January 1980. RFC 761.

[43] R. Ricci, E. Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[44] T. Schlossnagle. Monitoring in a DevOps world. *Communications of the ACM*, 61(3):58–61, Feb. 2018.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[46] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, Mississauga, Ontario, Canada, 1999. IBM Press.

[47] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998. Springer-Verlag.

[48] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, Banff, Alberta, Canada, 2001. ACM.

[49] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Broomfield, CO, 2014. USENIX Association.

[50] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 293–306, Hollywood, CA, USA, 2012. USENIX Association.

[51] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Conference on Hot Topics in System Dependability*, HotDep '05, Yokohama, Japan, 2005. USENIX Association.

REPT: Reverse Debugging of Failures in Deployed Software

Weidong Cui[1], Xinyang Ge[1], Baris Kasikci[2], Ben Niu[1], Upamanyu Sharma[2], Ruoyu Wang[3], and Insu Yun[4]

[1]Microsoft Research
[2]University of Michigan
[3]Arizona State University
[4]Georgia Institute of Technology

Abstract

Debugging software failures in deployed systems is important because they impact real users and customers. However, debugging such failures is notoriously hard in practice because developers have to rely on limited information such as memory dumps. The execution history is usually unavailable because high-fidelity program tracing is not affordable in deployed systems.

In this paper, we present REPT, a practical system that enables *reverse debugging* of software failures in deployed systems. REPT reconstructs the execution history with high fidelity by combining online lightweight hardware tracing of a program's control flow with offline binary analysis that recovers its data flow. It is seemingly impossible to recover data values thousands of instructions before the failure due to information loss and concurrent execution. REPT tackles these challenges by constructing a partial execution order based on timestamps logged by hardware and iteratively performing forward and backward execution with error correction.

We design and implement REPT, deploy it on Microsoft Windows, and integrate it into WinDbg. We evaluate REPT on 16 real-world bugs and show that it can recover data values accurately (92% on average) and efficiently (in less than 20 seconds) for these bugs. We also show that it enables effective reverse debugging for 14 bugs.

1 Introduction

Software failures in deployed systems are unavoidable and debugging such failures is crucial because they impact real users and customers. It is well known that execution logs are helpful for debugging [28], but nobody wants to pay a high performance overhead for always-on logging/tracing when most logs or traces would be discarded for normal runs. As a result, only a memory dump is captured upon failures in deployed software to enable post-mortem diagnosis.

Alas, it is challenging for developers to debug memory dumps due to limited information. The result is that a significant fraction of bugs is left unfixed [32,59]. Those that get fixed can take weeks in certain cases [32].

To make matters worse, streamlined software processes call for short release cycles [53], which limits the extent of in-house testing prior to software release. Frequent releases increase the dependency on debugging failures reported from deployed software, because these failure occurrences become the only way to detect certain bugs. Frequent releases also increase the demand for quickly resolving bugs to meet short release deadlines.

There exists a rich literature on debugging failures, which can roughly be classified into two categories:

(1) *Automatic root cause diagnosis* [16, 37–41, 61] attempts to automatically determine the culprit statements that cause a program to fail. Due to various limitations (e.g., requiring code modification [37, 40, 41], inability to handle complex software efficiently [37, 61], or being limited to a subset of failures [37, 39]), none of these systems are deployed in practice. Moreover, even though root cause diagnosis can help a developer determine the reasons behind a failure, developers often require a deeper understanding of the conditions and the state leading to a failure to fix a bug, which these systems do not provide.

(2) *Failure reproduction* for debugging attempts to enable developers to examine program inputs and state that lead to failures. Exhaustive testing techniques such as symbolic execution [22] and model checking [21, 58], or state-space exploration [51] can be used to determine inputs and state that lead to a failure for the purpose

of debugging. Unfortunately, these techniques require heavyweight runtime monitoring [26]. Another popular technique for reproducing failures is record/replay systems [46, 48, 50, 52, 56] that record program executions that can later be replayed to debug failures. This is also known as *reverse debugging* [31, 55] or *time-travel debugging* [44]. On the plus side, reverse debugging allows a developer to go back and forth in a failed execution to examine a program's state (i.e., control and data flow) to truly understand the bug and devise a fix. On the other hand, record/replay systems incur prohibitive overhead (up to 200% for the state-of-the-art system [56]) in multithreaded programs running on multiple cores, making them impractical for use in deployed systems.

Due to the limitations of existing techniques, major software vendors including Apple [17], Google [33], and Microsoft [30] as well as open-source systems such as Ubuntu [54] operate error reporting services to collect data about failures in deployed software and analyze them. To our knowledge, even the most advanced bug diagnosis system deployed in production, namely RETracer [27], is only able to *triage* failures caused by access violations.

To solve the challenge of debugging software failures in deployed systems, we argue that we need a *practical* solution that enables reverse debugging of such failures. To be practical, the solution must (1) impose a very low runtime performance overhead when running on a deployed system, (2) should be able to recover the execution history accurately and efficiently, (3) work with unmodified source code/binary, (4) apply to broad classes of bugs (e.g., concurrency bugs).

In this paper, we present REPT[1], a practical solution for reverse debugging of software failures in deployed systems. There are two key ideas behind REPT. First, REPT leverages *hardware* tracing to record a program's control flow with low performance overhead. Second, REPT uses a novel binary analysis technique to recover data flow information based on the logged control flow information and the data values saved in a memory dump. Consequently, REPT enables reverse debugging by combining the logged control flow and the recovered data flow.

The main challenge faced by REPT is how to *accurately* and *efficiently* recover data values based on the logged control flow and the data values saved in the memory dump. To be *accurate*, REPT must be able to correctly recover a significant fraction of data values in the execution history. To be *efficient*, REPT must incur low runtime monitoring overhead and should finish its analysis within minutes. To solve this challenge, we introduce a new binary analysis approach that combines *forward* and *backward* execution to *iteratively* emulate instructions and recover data values. REPT uses the following two new techniques for its analysis:

First, we design an error correction scheme to detect and correct value conflicts that are introduced by memory writes to unknown addresses. When emulating a memory write instruction, it is too conservative to mark all memory values as unknown if the destination address is unknown. Instead, REPT leaves memory untouched and relies on detecting a conflict later caused by stale values in the destination memory. Unlike previous solutions that use expensive hypothesis tests to decide memory aliases [57], the error correction scheme enables REPT to run its iterative analysis efficiently.

Second, we leverage the timing information provided by modern hardware to determine the order of non-deterministic events such as races across multiple threads. Non-determinism has been a long-standing challenge that hinders the ability of existing record/replay systems to achieve high accuracy with low overhead. REPT can identify the order of accesses to the same memory location in most cases by using fine-grained timestamps that modern hardware provides. When the timing information is not enough, REPT restricts the use of memory accesses whose order cannot be inferred. This stops their values from negatively affecting the recovery of other data.

We implement REPT in two components. The online tracing component is a driver that controls Intel Processor Trace (PT) [36], and has been deployed on hundreds of millions of machines as part of Microsoft Windows. The offline binary analysis component is a loadable library that is integrated into WinDbg [45]. We also enhance Windows Error Reporting (WER) service [30] to control hardware tracing on deployed systems.

To measure the effectiveness and efficiency of REPT, we evaluate it on 16 real-world bugs in software such as Chrome, Apache, PHP, and Python. Our experiments show that REPT can enable effective reverse debugging for 14 of them, including 2 concurrency bugs. We evaluate REPT's data recovery accuracy by comparing its recovered data values with those logged by Time Travel Debugging (TTD) [44], a slow but precise record/replay tool. Our experiments show that REPT can achieve an average accuracy of 92% and finish its analysis in less than 20 seconds for these bugs.

[1] REPT stands for Reverse Execution with Processor Trace and reads as "repeat."

2 Overview

2.1 Problem Statement

The overarching goal of REPT is to enable reverse debugging of failures in deployed software with low runtime overhead. REPT realizes reverse debugging in two steps. (1) REPT uses hardware support to log the control flow and timing information of a program's execution. When a failure occurs, REPT saves an enriched memory dump including both the final program state and the additionally recorded control flow and timing information before the failure. (2) REPT uses a new offline binary analysis technique to recover data values in the execution history based on the enriched memory dump.

REPT needs to recover data values because there is no existing hardware support for efficiently logging all data values of a program's execution. However, there exist hardware features such as Intel PT [36] and ARM Embedded Trace Macrocell [18] that can efficiently log the control flow and timing information.

2.2 Design Choices

When designing REPT, we make three design choices.

Memory Dump Only vs. Online Data Capture: We choose to only rely on the data in a memory dump rather than logging more data during execution to minimize the performance overhead for deployed systems. Furthermore, to do online data capture, we would need to modify the operating system or programs because there is no existing hardware support for that. We choose not to do it to minimize intrusiveness.

Binary vs. Source: We choose to do the analysis at the binary level instead of at the source code level for three reasons. First, by performing analysis at the instruction level, REPT is essentially agnostic to programming languages and compilers. This allows REPT to support native languages (e.g., C/C++) as well as managed languages (e.g., C#). Second, today's applications often consist of multiple modules/libraries from different vendors, and not all source code may be available for analysis [25]. Third, the mapping between the source code and binary instructions is not straightforward due to compiler optimizations and the use of temporary variables, thus converting source-level analysis result back to the binary-level presents a non-trivial challenge.

Concrete vs. Symbolic: One popular approach to reconstructing executions is symbolic execution. In symbolic execution, a program is *executed* with *symbolic* inputs of *unconstrained* values (e.g., a Boolean can initially take any of the true or false values) as opposed to concrete ones. As the program executes, symbolic execution gathers constraints on symbolic values. Whenever an event of interest occurs (e.g., a failure), symbolic execution uses a constraint solver to determine the program inputs that would have led to that failure. Conceptually, symbolic execution may help with recovering data values. We could treat operands such as registers and memory locations referenced by each instruction as variables, and generate constraints among these variables based on the semantics of the instructions. However, given a long execution trace, the constraints gathered on the variables may grow too large (particularly when memory locations are made symbolic) to solve within a reasonable amount of time for even state-of-the-art constraint solvers. Therefore, we choose to do concrete execution instead of symbolic execution. REPT keeps concrete values for registers and memory locations at each position in the instruction sequence and analyzes each instruction to recover concrete values of its operands.

2.3 Challenges

To enable reverse debugging, REPT faces three challenges when recovering register and memory values in the execution history.

2.3.1 Irreversible Instructions

This first challenge for REPT *is handling irreversible instructions*. If every instruction is *reversible* (i.e., the program state before an instruction's execution can be fully determined based on the program state after its execution), then the design of REPT would be straightforward: invert each instruction's semantics and recover data values at each position in the instruction sequence. However, many instructions are *irreversible* (e.g., `xor rax,rax`) and thus information destroying. We solve this challenge by using forward execution to recover values that cannot be recovered in backward execution.

2.3.2 Missing Memory Writes

The second challenge for REPT is *handling memory writes to unknown addresses*. Most memory addresses cannot be determined statically. Since the analysis may not fully recover data values due to irreversible instructions, REPT may not know the destination of a memory write during its analysis. When this happens, one option is to assume that values at *all* memory locations become unknown. This is too conservative because it may cause the analysis to miss many data values that are actually recoverable. If REPT chooses to ignore the memory

write, the analysis will leave an invalid value at the memory location, which may propagate into other registers or memory locations. We solve this challenge by using error correction.

2.3.3 Concurrent Memory Writes

The third challenge for REPT is *correctly identifying the order of shared memory accesses*. In the presence of multiple instruction sequences from different threads, it may not be possible to infer the execution order of concurrent memory accesses despite timestamps provided by hardware. REPT needs to properly handle these memory accesses, otherwise it may infer wrong values for these memory locations. We solve this challenge by restricting in the analysis the use of data values recovered from concurrent memory accesses.

3 Design

In this section, we describe the design of REPT by focusing on how it solves the three key technical challenges discussed in the previous section.

For brevity, we define an instruction sequence as $I = \{I_i | i = 1, 2, ..., n\}$ where I_i represents the i-th instruction executed in the sequence. We assume that the memory dump is available after the n-th instruction's execution. We define a program's state, S, as a collection of all data values in registers and memory locations. We define S_i as the program state after the i-th instruction is executed. Therefore, S_0 represents the program state before the first instruction I_1 is executed, and S_n represents the program state stored in the memory dump. We define a state S_i as *complete* if all the register and memory values are known. We define an instruction I_i as *reversible* if, given a complete state S_i, we can recover S_{i-1} completely; otherwise we say the instruction is *irreversible*. The design of REPT is not limited to a specific architecture, however, in the rest of the paper, we use x86-64 instructions in our examples.

In the rest of this section, we present the design of REPT progressively by describing how it handles increasingly more complex and realistic scenarios.

- A single instruction sequence with only **reversible** instructions (Section 3.1).
- A single instruction sequence with **irreversible** instructions but without memory accesses (Section 3.2).
- A single instruction sequence with irreversible instructions and **with memory accesses** (Section 3.3).
- **Multiple** instruction sequences with irreversible instructions and with memory accesses (Section 3.4).

3.1 Instruction Reversal

REPT's first mechanism assumes that the input is a single instruction sequence with only reversible instructions. Since every instruction is reversible, REPT can reverse the effects of each instruction to completely recover the initial program state from the end of the instruction sequence to the beginning. For instance, if the instruction sequence has a single instruction I_1 = `add rax,rbx` and S_1 = {`rax`=3, `rbx`=1}, then the analysis can recover S_0 = {`rax`=2, `rbx`=1}.

3.2 Irreversible Instruction Handling

REPT's second mechanism assumes that there is a single instruction sequence with irreversible instructions, but the sequence does not include any memory access. In practice, most instructions are irreversible. For instance, `xor rbx,rbx` is irreversible, because `rbx`'s value before the instruction is executed cannot be recovered simply based on this instruction's semantics and `rbx`'s value after the instruction is executed. Therefore, the straightforward backward analysis for reversible instructions is not applicable in general.

The key idea for recovering a *destroyed* value is to infer it in a *forward* analysis. As long as the destroyed value is derived from some other registers and memory locations, and their values are available, we can use these values to recover the destroyed value. Extending this idea, our basic solution is to iteratively perform backward and forward analysis to recover data values until no new values are recovered.

Conceptually, given the instruction sequence I and the final state S_n, we first mark all register values as *unknown* in program states from S_0 to S_{n-1}. Then we do backward analysis to recover program states from S_{n-1} to S_0. After this step, we perform forward analysis to update program states from S_0 to S_{n-1}. We repeat these steps until a *fixed point* is reached: i.e., no state is updated in a backward or forward analysis. When we update a program state, we only change a register's value from unknown to an inferred value. Crucially, this analysis will not produce conflicting inferred values because all the initial values are correct and no step in the analysis can introduce a wrong value based on correct values. This also guarantees that the iterative analysis will converge.

We show an example of handling irreversible instructions in Figure 1. The instruction sequence has three instructions, and two of them are irreversible. Since we do

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑{rax=?, rbx=?} →	↓	↑{rax=2, rbx=?}
I_1	mov rbx, 1	S_1	↑{**rax=?**, rbx=?}	↓{rax=?, **rbx=1**}	↑{**rax=2**, rbx=1}
I_2	add rax, rbx	S_2	↑{rax=3, **rbx=?**}	↓{**rax=3**, rbx=1}	↑{rax=3, rbx=1}
I_3	xor rbx, rbx	S_3	↑{rax=3, rbx=0}	↓{rax=3, rbx=0} →	↑

Figure 1: This example shows how REPT's iterative analysis recovers register values in the presence of irreversible instructions. We use "?" to represent "unknown". Key updates during the analysis are marked in bold face.

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑{rax=?, rbx=?, [g]=3} →		↑{rax=?, rbx=?, [g]=2}
I_1	lea rbx, [g]	S_1	↑{rax=?, rbx=?, [g]=3}	↓{rax=?, rbx=g, [g]=3}	↑{rax=?, rbx=g, [g]=2}
I_2	mov rax, 1	S_2	↑{rax=?, rbx=?, [g]=3}	↓{rax=1, rbx=g, [g]=3}	↑{rax=1, rbx=g, **[g]=2**}
I_3	add rax, [rbx]	S_3	↑{rax=3, rbx=?, **[g]=3**}	↓{**rax=3**, rbx=g, [g]=3}	↑{rax=3, rbx=g, **[g]=?**}
I_4	mov [rbx], rax	S_4	↑{rax=3, rbx=?, [g]=3}	↓{rax=3, rbx=g, [g]=3}	↑{rax=3, rbx=g, [g]=3}
I_5	xor rbx, rbx	S_5	↑{rax=3, rbx=0, [g]=3}	↓{rax=3, rbx=0, [g]=3} →	

Figure 2: This example shows how REPT's iterative analysis recovers register and memory values when there exist irreversible instructions with memory accesses. We use "?" to represent "unknown", and use "g" to represent the memory address of a global variable. Some values are in bold-face because they represent key updates in the analysis. We skip the fourth iteration which will recover [g]'s value to be 2 due to the space constraint.

not have instructions before the first one, we do not expect to recover rbx in S_0. There are three points that are worth noting in this example. First, we recover rbx's value in S_1 based on the forward analysis in the second iteration. Second, we keep rax's value of 3 in S_2 in the second iteration of forward analysis even though rax's value is unknown in S_1. Third, we recover rax's value of 2 in S_1 in the last iteration of backward analysis.

3.3 Recovering Memory Writes

REPT's third mechanism assumes that there is a single instruction sequence with irreversible instructions and with memory accesses. In practice, there are always instructions that access memory. Unlike registers that can be statically identified from instructions, the address of a memory access may not always be known. For a memory write instruction whose destination is unknown, we cannot correctly update the value for the destination memory. A missing update may introduce an obsolete value, which would negatively impact subsequent analysis. A conservative approach that marks all memory as unknown upon a missing memory write would lead to an unnecessary and unacceptable information loss.

Our key insight for solving the missing memory write problem is to use *error correction*. The intuition behind REPT is to keep using the memory values that are possibly valid to infer other values, and to correct the values later if the values turn out to be invalid based on conflicts. Before describing REPT's error correction algorithm, we first use an example to explain the high-level idea.

The example in Figure 2 has five instructions. There are three key updates as marked in bold face. In the first iteration of the backward analysis, since we do not know rbx's value in S_4, we do not change the value at the address g. In the second iteration of the forward analysis, there is a conflict for rax in S_3. The original value is 3, but the newly inferred value would be 4 (rax + [g] = 1 + 3 = 4). Our analysis keeps the original value of 3 because it was inferred from the final program state which we assume is correct. In the third iteration of the backward analysis, based on rax's value before and after the instruction I_3, we can recover [g]'s value to be 2.

Next, we describe the algorithm that REPT uses to recover missing memory writes. We first introduce the data inference graph in Section 3.3.1, and then explain how we use the graph to detect and correct errors caused by missing memory writes in Section 3.3.2.

3.3.1 Data Inference Graph

When performing the backward and forward analysis, REPT maintains a *data inference graph*. The data inference graph is different from a traditional data flow graph in the sense that it tracks how a data value is inferred in either forward or backward directions while a data flow graph tracks the program's data flow in just one direction.

An example data inference graph is shown in Figure 3. In this example, we use rcx to recover [rax], and then use the latter to recover rbx. Here we assume that rax's value is not changed between I_1 and I_n.

A node in the data inference graph represents a register or a memory location that is accessed in an executed instruction. A node is called a *use* node if its correspond-

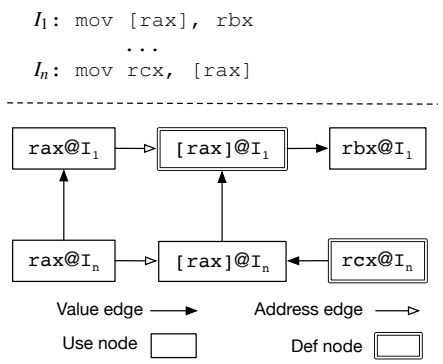


Figure 3: An example data inference graph in REPT. The graph indicates that REPT uses `rcx`@I_n to recover `[rax]`@I_n, which is further used to recover `[rax]`@I_1 and subsequently `rbx`@I_1.

ing register or memory location is for read. Similarly, a node is called a *def* node if it is for write. For instance, `rbx`@I_1 is a use node, and `rcx`@I_n is a def node. If a register or memory location is accessed for both read and write in a single instruction, we create two nodes for it: one use node, and one def node. Finally, REPT treats data in the memory dump as use nodes because their values can be propagated backwards like other use nodes.

There are two kinds of directional edges in the data inference graph: *value edges* and *address edges*. A value edge from node A to node B means that REPT uses A's value to infer B's value. An address edge from A to B means that A's value is used to compute B's address. For instance, the edge from `rcx`@I_n to `[rax]`@I_n is a value edge, and the edge from `rax`@I_n to `[rax]`@I_n is an address edge. To get or set the value of a memory location, its address must be known. When setting a memory node's value, besides value edges, REPT adds address edges from register nodes that are used to compute the address of the memory node. A memory node can have multiple incoming address edges (e.g., a base register and an index register are used together to specify the address).

There are two types of value edges. In the first type of value edges, the connected nodes are from the same instruction and we call them *horizontal* edges. Specifically, in the backward analysis, if a def node's value is known and can be used to infer the value of a use node in the same instruction, we recover the use node's value and add a horizontal edge between the two nodes. Similarly, in the forward analysis, if a use node's value is known and can be used to infer the value of a def node in the same instruction, we recover the def node's value and add a horizontal edge between the nodes as well. It is worth noting that a node may have multiple horizontal incoming value edges. For instance, given `add rax,rbx`, the def node of `rax` can have two incoming value edges from the use nodes of `rax` and `rbx`.

In the second type of value edges, the connected nodes are from different instructions, but they correspond to the same register or memory location. Such value edges are referred to as *vertical* edges. Intuitively, nodes connected via vertical edges belong to the same def-use chain (i.e., a single def with all its reaching uses). In the backward analysis, we recover values from a use node to the preceding use node or the def node along the def-use chain, and add vertical edges in between. Similarly, in the forward analysis, we recover values from a def or use node to its subsequent use node along the def-use chain and add corresponding vertical edges as well. In other words, a def node's value can only be propagated forwardly while a use node's value can be propagated on both directions.

For every node in the data inference graph, REPT also maintains a *dereference level* to aid in error correction (Section 3.3.2). Specifically, all use nodes of values in the memory dump have a dereference level of 0. For any other node, REPT determines its dereference level in three steps: (1) for all incoming value edges, find the maximum dereference level of the source nodes as D_1; (2) for all incoming address edges, find the maximum dereference level of the source nodes as D_2; (3) pick the larger value between D_1 and D_2+1 as the target node's dereference level. We can see that the dereference level actually measures the maximum number of address edges from a value stored in the memory dump to the given node. A node's dereference level reflects the confidence level for its value since data inference errors come from memory due to missing memory writes. A higher dereference level means a lower confidence level.

3.3.2 Error Correction

During the iterative backward and forward analysis, REPT continuously updates the data inference graph and detects and corrects inconsistencies. There are two kinds of inconsistencies: *value* conflict and *edge* conflict. A value conflict happens when an inferred value does not match the existing value. An edge conflict happens when a newly identified def node of a memory location *breaks* the previously assumed def-use relationship between two nodes connected through a vertical edge. Consider the example in Figure 3. If REPT detects another write to the same memory location specified by `rax` between I_1 and I_n, this memory write will cause a conflict on the

vertical edge between `[rax]`@I_n and `[rax]`@I_1.

When REPT detects a conflict, it stops the analysis of the current instruction, identifies the invalid node, then runs the invalidation process. For both types of conflicts, the invalidation process starts with an initial node. In the case of edge conflicts, the initial node is the target node of the broken vertical edge as it no longer belongs to the same def-use chain. In the case of value conflicts, REPT checks if the dereference level of the node of the newly inferred value is less than or equal to that of the node of the existing value (this means a higher or equal confidence for the new value). If so, REPT picks the node of the existing value as the initial node for invalidation. Otherwise, REPT discards the newly inferred value and moves on to the next instruction.

If REPT identifies an initial node for invalidation, it first processes each of its outgoing value and address edges. For a value edge, the target node is marked as unknown. For an address edge, the target node is deleted from the data inference graph since its address becomes unknown and consequently such a def or use on that memory location may no longer exist. Then REPT recursively applies the invalidation process to these target nodes. It is worth noting that the data inference graph is guaranteed *not* to have cycles, because REPT adds a node and edges into the graph only when the node's value is inferred for the first time.

To ensure convergence of the analysis, REPT maintains a *blacklist* of invalidated values for each node. Every time a node is invalidated, its value is added to its blacklist. Once a value is in a node's blacklist, the node cannot take that value any more. This ensures that the iterative analysis process will not enter the conflicting state again and consequently guarantees that the algorithm will eventually converge. However, a correct value can be incorrectly blacklisted for a node if it has a lower confidence level than another incorrect value. This leads to the problem that a value is recoverable but cannot be recovered due to the use of the blacklist. We choose to keep the blacklists to prioritize the convergence of the analysis over the improvement in data recovery.

3.4 Handling Concurrency

When we face multiple instruction sequences executed simultaneously on multiple cores, the problem is seemingly intractable because, without a perfect order of the executed instructions, there could be a large number of ways to order those instructions. We have two insights for tackling this challenge. First, we leverage the timing information logged by hardware tracing to construct a partial order of instructions executed in different threads. Second, we recognize that memory writes are the *only* operations whose orders may affect data recovery.

With timestamps inserted in an instruction sequence, we refer to the instructions between two timestamps as an instruction *subsequence*. We refer to the two timestamps as the start and end time of the subsequence. Given two instruction subsequences from two different instruction sequences, we infer their relative execution order based on their start and end times. If one subsequence's end time is before another subsequence's start time, we say the first subsequence is executed *before* the other subsequence. Otherwise, we say their order cannot be inferred, and the two subsequences are *concurrent*. Note that the order of two subsequences in the same instruction sequence can always be determined based on their positions in the instruction sequence. We say two instructions are *concurrent* if the instruction subsequences they belong to are concurrent. We say two memory accesses are *concurrent* if the corresponding memory access instructions are concurrent.

Given multiple instruction sequences executed simultaneously on multiple cores, REPT first divides them into subsequences, then merges them into a *single conceptual* instruction sequence based on the inferred orders. For two subsequences whose order cannot be inferred, REPT arbitrarily inserts one before the other in the newly constructed sequence. A natural question is whether the data recovery is affected by this arbitrary choice of ordering two concurrent subsequences. Obviously, if we change the order of two subsequences that have concurrent memory accesses to the same location and one of them is write, we may get different values for the memory location. On the other hand, if concurrent subsequences do *not* have any concurrent memory write to the same location, it does not matter in which order REPT places them into the merged instruction sequence.

Since we cannot tell the order of concurrent instruction subsequences, our goal is to *eliminate* the impact of their ambiguous order on data recovery. Specifically, during the iterative analysis, for every memory access (regardless of read or write), REPT detects if it has a concurrent memory write to the same location. If so, REPT takes the following steps to limit the use of the memory access in the data inference graph. First, REPT removes all vertical edges of the node representing the memory access and invalidates the target nodes of outgoing vertical edges. Then, REPT labels the memory access node so that it will not be used in vertical edges. This is because REPT does not know if the memory access happens before or after the concurrent memory write to the same

location. However, REPT still allows horizontal value edges to infer this node's value.

A remaining question is whether picking an arbitrary order for concurrent instruction subsequences would affect the detection of concurrent memory writes to the same location. Our observation is that REPT's analysis works as long as there are no two separate concurrent writes such that one affects the inference of another's destination. We acknowledge that this possibility exists and depends on the granularity of timing information. Given the timestamp granularity supported by modern hardware, we deem this as a rare case in practice [39].

4 Implementation

In this section, we first describe the implementation details of REPT's online hardware tracing and offline binary analysis. Then we describe its deployment.

4.1 Online Hardware Tracing

REPT leverages Intel Processor Trace (PT) to log control-flow and timing information of a program's execution. Intel PT became available when the Broadwell architecture was released in 2014. Intel PT supports various program tracing modes, and REPT currently uses the *per-thread circular* buffer mode to trace user-space execution of all threads within a process. REPT supports configuring the circular buffer size and the granularity of timestamps. We do not configure Intel PT to do whole-execution tracing because that would introduce performance overhead due to frequent interrupts (when the trace buffer gets full) and I/O workload (when the buffer is written to some persistent storage). When a traced process fails, its final state and the recorded Intel PT traces are saved in a single memory dump.

4.2 Offline Binary Analysis

REPT takes a memory dump with Intel PT trace as input, and outputs the recovered execution history of each thread. At first, REPT parses the trace to reconstruct the control flow. Parsing an Intel PT trace requires that the binary code in the dump is the same as the code that was executed when the trace is collected. Therefore, REPT supports jitted code as long as the code was not modified since its execution was logged in the circular trace buffer. Next, REPT converts native instructions into an intermediate representation (IR) that specifies opcodes and operands, and conducts the forward and backward analysis until it converges.

In addition to the final program state and constants, REPT can leverage control dependencies to recover data. For instance, if a conditional branch is executed only if a register's value is 0, then REPT can infer the register's value once it observes that the branch is taken.

Programs invoke system calls to request operating system services, and the operating system may modify certain register and memory values in the process as a response. Upon a system call, REPT will mark all volatile registers as unknown based on the calling convention. REPT currently does not handle memory writes by the kernel, but instead treats those in the same way as missing memory writes and relies on the error correction mechanism to detect and resolve conflicts. We acknowledge that semantic-aware handling of system calls can be done with more engineering effort to help improve the data recovery, but we leave it to future work.

4.3 Deployment

We implement REPT in two components and deploy it into the ecosystem of Microsoft Windows for program tracing, failure reporting, and debugging.

First, we implement the online hardware tracing component as a driver of 8.5K lines of C code. It is responsible for controlling tracing of a target process and capturing the trace in a memory dump when the monitored process fails. We also modify the Windows kernel to support per-thread tracing by swapping the trace buffers upon context switch.

Second, we implement REPT's offline binary analysis and reverse debugging as a library of 100K lines of C++ code, and integrate it into WinDbg [45]. We also implement common debugging functionalities such as code and data breakpoints to facilitate the debugging process.

We enhance the Windows Error Reporting (WER) service [30] to support REPT. Specifically, developers can request Intel PT enriched memory dumps on WER. Then WER selects user machines to trace the targeted program. When a traced program causes a failure, a memory dump with Intel PT trace is captured and sent back to WER. Finally, developers can load the enriched memory dump in WinDbg to do reverse debugging.

5 Evaluation

In this section, we evaluate REPT to answer the following four questions: (1) How accurately can REPT recover data values? (2) How efficiently can REPT recover data values? (3) How effectively can REPT be used to debug failures? (4) What is the deployment status? Next,

Program-BugId	Bug Type	MP	SS
Apache-24483	NULL pointer deref [1]	No	Yes
Apache-39722	NULL pointer deref [2]	No	Yes
Apache-60324	Integer overflow [3]	No	Yes
Nasm-2004-1287	Stack buffer overrun [4]	No	No
PHP-2007-1001	Integer overflow [5]	No	Yes
PHP-2012-2386	Integer overflow [6]	No	No
PHP-74194	Type confusion [7]	No	No
PHP-76041	NULL pointer deref [8]	No	Yes
PuTTY-2016-2563	Stack buffer overrun [9]	No	No
Python-2007-4965	Integer overflow [10]	No	Yes
Python-28322	Type confusion [11]	No	No
Chrome-784183	Integer overflow [12]	No	No
Pbzip2	Use-after-free [29]	Yes	No
Python-31530	Race [13]	Yes	No
Chrome-776677	Race [14]	Yes	No
LibreOffice-88914	Deadlock [15]	Yes	No

Table 1: Software bugs used in our experiments. MP means that the defect and failure threads are different. SS means that the defect is on the same stack as the failure.

we present our experimental setup and describe our experimental results to answer these questions.

We evaluate REPT on failures caused by 16 real-world bugs listed in Table 1. All of these bugs are from open-source software. We focus on open-source software for independent reproducibility. The main constraint that limits us from evaluating REPT on more bugs is that we need to reproduce bugs in open-source software on Microsoft Windows. When reproducing bugs, we try to pick bugs that are from a diverse set of widely-used real-world systems (e.g., Apache, Python, Chrome and PHP) and from a wide spectrum of bug types (e.g., NULL pointer dereference, race, type confusion, use-after-free, integer overflow, and buffer overflow).

In our experiments, we configure Intel PT to use a circular buffer of 256K bytes per thread and turn on the most fine-grained timestamp logging (i.e., TSCEn=1, CYCEn=1, CycThresh=0 and MTCFreq=0; see [36] for more details).

5.1 Accuracy

To evaluate the accuracy of REPT's data recovery, we need to obtain the ground truth. We use Time Travel Debugging (TTD) [44], a slow but precise record/replay tool, to log both control and data flow of a program's execution. With the fully recorded execution, we create inputs to REPT and check the correctness of its output. To evaluate the accuracy of REPT in handling multiple concurrent instruction sequences, we modify TTD to generate the timing information as an approximation to timestamps generated by Intel PT. Finally, we stress test REPT on a highly concurrent program and report how well the timestamps provided by Intel PT can order shared memory accesses under extreme cases.

Program-BugId	# Insts	Cor	Unk	Inc
Apache-24483	49	96.72%	1.64%	1.64%
Apache-39722	1,644	99.30%	0.70%	0.00%
Apache-60324	672	96.47%	1.83%	1.70%
Nasm-2004-1287	67,726	95.95%	3.70%	0.35%
PHP-2007-1001	54,475	99.08%	0.90%	0.02%
PHP-2012-2386	43,813	71.55%	25.40%	3.05%
PHP-74194	78,103	90.88%	7.82%	1.30%
PHP-76041	115	94.96%	3.60%	1.44%
PuTTY-2016-2563	677	99.55%	0.45%	0.00%
Python-2007-4965	1,043	95.04%	4.09%	0.87%
Python-28322	1,062	90.85%	8.60%	0.55%

Table 2: REPT's accuracy on a single instruction sequence. Cor, Unk and Inc represent the percentage of correct, unknown, and incorrect register uses.

5.1.1 Single-Thread Accuracy

In this experiment, we first use TTD to record the execution where each bug is triggered. Then, we replay the recorded execution to construct an instruction sequence without the timing information for the failure thread. Next, we run REPT on the constructed instruction sequence and the final program state provided by the replay engine. Finally, we compare the recovered data values with the data values returned by the replay engine.

When we compare the data values, we only check *register uses* (i.e., a register used as a source operand or the address of a destination memory operand). We do not check *defs* (i.e., a destination operand) because we want to avoid double counting. For instance, given `mov rax,rcx`, both `rax` and `rcx` will be correct or incorrect at the same time. When computing the data recovery accuracy, we do not need to count both of them. We do not check *memory uses* (i.e., a memory used as a source operand) because memory values are usually read into registers before they take on any operations. We analyze the trace of the 16 bugs and find that the destination is a register for 95% of memory reads. Therefore, we can count the uses of these registers to measure the accuracy.

We present our accuracy measurements in Table 2. Column 2 describes the number of instructions executed from the program defect to the program failure. We identify the location of a program defect based on the bug fix. For instance, Apache-24483 is a NULL pointer dereference bug, and its defect is where the NULL pointer check

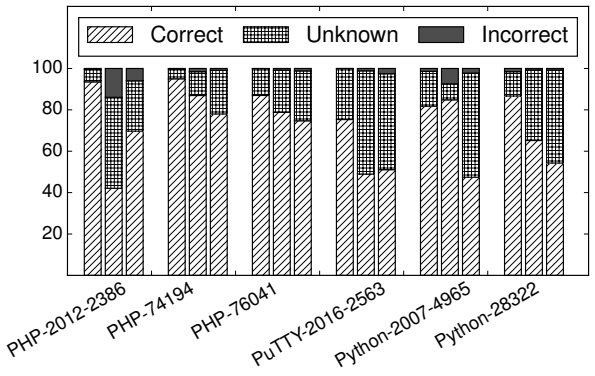


Figure 4: REPT's accuracy on different instruction sequence sizes. For each bug, we limit REPT to analyze 1M instructions, and depict the accuracy for 10K, 100K and 1M instructions away from failure, from left to right.

is added in the bug fix. The rest of three columns show the percentage of correct, unknown and incorrect register uses recovered by REPT in the instruction sequence from the defect to the failure.

We can see that REPT achieves a high accuracy. In most cases, the percentage of correct register uses is above 90% for tens of thousands of instructions; the percentage is still above 80% within 162,208 instructions for the Python-31530 bug. PHP-2012-2386 is an outlier case with the lowest accuracy. This particular bug involves a large number of memory allocation operations right before the program failure. Unfortunately, memory allocation operations are hard to reverse because the metadata information (i.e., chunk sizes) may be completely overwritten by reallocations, resulting in a large percentage of unknowns. We could not obtain the ground truth for Chrome-781483 because TTD does not support Chrome.

We also evaluate how the data recovery accuracy changes as the trace grows. We use instruction sequence sizes of 10K, 100K and 1M, and evaluate 6 bugs, because others have short execution histories. The results are summarized in Figure 4. Overall, the accuracy decreases as the number of instructions increases, and the rate of decrease depends on the program and the workload. It is worth noting that the accuracy does not decrease monotonically as the number of instructions increases. This is expected because REPT's accuracy depends on a program's behavior. For instance, PHP-2012-2386 has the accuracy drop in the case of 100K instructions because these instructions have a large number of memory allocation operations which are hard to reverse.

5.1.2 Multiple-Thread Accuracy

To evaluate REPT's analysis on multiple concurrent executions, we need to emulate the timing information in addition to the control flow from TTD. Currently, TTD supports record and replay of multithreaded programs running on multiple cores by logging timestamps at each system call and synchronization operation (e.g., `cmpxchg`). We extend TTD to log timestamps periodically in a manner similar to Intel PT during recording. When constructing an instruction sequence, we insert TTD's timestamps into the sequence accordingly. We acknowledge that such an approach may not perfectly reflect a multithreaded program's actual behavior on a bare metal machine.We conduct this experiment and report the results as our best estimation of REPT's accuracy for multithreaded programs.

We evaluate REPT on two race condition bugs, Pbzip2 and Python-31530. We do not evaluate Chrome-776677 or LibreOffice-88914 because REPT does not work for them (see Section 5.3). We measure the accuracy on the instructions executed on all threads from the defect to the failure. For Pbzip2, there are 12,496 instructions, and the correct/unknown/incorrect percentages are 95.33%, 4.36%, and 0.31%. For Python-31530, there are 511,289 instructions, and the corresponding percentages are 75.72%, 24.14%, and 0.14%. We attribute the lower accuracy on Python-31530 to the large number of instructions elapsed between the defect and the failure.

Finally, we evaluate how well REPT can use fine-grained timestamps from Intel PT to order memory accesses. We use Racey [34], a stress-testing benchmark that has extremely frequent data races—each thread races with other threads to constantly read/write a shared array for updating a signature. We run Racey with 8 threads for 1000 iterations and instrument it to save the addresses of memory accesses to the shared array. To minimize the instrumentation's impact on timing, we store the memory addresses to a pre-allocated buffer. We measure the fraction of memory accesses that have concurrent memory writes to the same location. We find that 5.5% of accesses to the shared array have concurrent memory writes. Given Racey is an extreme case of concurrent programs, we believe that the granularity of timestamps provided by Intel PT is sufficient for a majority of real-world programs.

5.2 Efficiency

Efficiency of REPT has two prongs, the performance overhead caused by Intel PT when a program is running, and REPT's offline analysis for data recovery. The for-

Program-BugId	# Iters	REPT (s)
Apache-24483	4	5.8
Apache-39722	5	3.0
Apache-60324	2	5.5
Chrome-784183	6	8.2
Nasm-2004-1287	10	18.6
Pbzip2	7	8.2
PHP-2007-1001	5	2.0
PHP-2012-2386	6	3.8
PHP-74194	7	6.3
PHP-76041	6	14.5
PuTTY-2016-2563	5	5.2
Python-2007-4965	12	10.5
Python-28322	18	17.5
Python-31530	6	10.6

Table 3: The number of iterations and the time of REPT's offline analysis.

mer is low and has been well studied. For instance, Figure 8 in [39] shows that the performance overhead with circular buffers and the timing information is below 2% for a range of applications. Furthermore, the deployment of REPT proves that its performance overhead is acceptable in practice, particularly when it is selectively turned on for a program on a user machine.

We test REPT's offline analysis on a machine running an x86-64 Windows 10 on an Intel Core i7-7700K 4.2GHZ Quad-Core CPU with 16GB RAM. In Table 3, we show the analysis time for the 14 bugs REPT can analyze. We can see that REPT finishes its analysis within 20 seconds for all the 14 bugs.

5.3 Effectiveness

To evaluate the effectiveness of REPT, we check if reverse debugging based on recovered data can be used to effectively diagnose a bug. To make this check objective, we say REPT is effective if the values of variables that are involved in the bug fix are correctly recovered. For all the 16 bugs listed in Table 1, REPT is effective for 14 bugs. REPT does not work for Chrome-776677 because the collected trace contains in-place code update for jitted code, which fails Intel PT trace parsing. REPT does not work for LibreOffice-88914, because this is a deadlock bug that triggers an infinite loop, which easily fills up the circular trace buffer and causes the program execution history before the loop to be lost. Out of those 14 bugs, we select three complicated ones to demonstrate the effectiveness of REPT.

Pbzip2. This is a use-after-free bug caused by a race condition. Pbzip2 is a parallel file (de)compressor based on bzip2. Specifically, it divides an input file into chunks of an equal size and spawns multiple child threads to process them in parallel. The main thread synchronizes with child threads using a mutex. Unfortunately, there is a race condition bug where the main thread may free the mutex before all child threads finish, causing the program to crash when a child thread dereferences a pointer field inside the freed mutex. With REPT, a developer can set a data breakpoint on the pointer field, and locate the instruction that overwrites the pointer field in the heap free operation on the main thread by going backwards along the execution.

Python-31530. This is a race condition bug in Python's implementation of its file objects. Python preloads the file content as an optimization for its file operations. To do so, Python allocates a buffer based on the given size `bufsize` and assigns it to a pointer field `f_buf` in the file object. Then, it reads the file content into the buffer, and finally updates another pointer field `f_bufend` so that it points to the end of the buffer (i.e., `f_bufend=f_buf+bufsize`). The race condition happens when two threads preload the file content simultaneously. Specifically, while a thread is reading file content into the buffer, another thread starts preloading and overwrites `f_buf` with a *smaller* buffer. Then, the original thread updates `f_bufend` based on the overwritten `f_buf` and the old `bufsize`, which makes `f_bufend` point to a location beyond the actually allocated buffer. This causes Python to crash when it attempts to read the data outside of the allocated buffer. With REPT, a developer can set data breakpoints on both `f_buf` and `f_bufend`. By going backwards along the reconstructed execution, the developer can see how the race condition bug overwrites `f_buf` and leads to an inconsistent `f_bufend`.

Chrome-784183. This is an integer overflow bug in a validation routine used for image snipping. The validation routine checks if the snipped area is within the original image. For example, given an image represented as a matrix of pixels, one can snip the image by choosing `y` rows from row `x`. The validation routine ensures `x+y` is not greater than the height of the original image. Unfortunately, the routine does not check if `x+y` overflows. Thus, the check is incorrectly passed when a large `y` causes an integer overflow. This results in the subsequent crash when Chrome attempts to access a pixel in the snipped area based on `y`. When the crash happens, the validation function has already returned and more than 500K instructions have been executed afterwards. With REPT, a developer can go back to the validation routine and single step through it to quickly pinpoint the actual arithmetic operation that overflows.

5.4 Deployment

We have received anecdotal stories from Microsoft developers in using REPT to successfully debug failures reported to WER [30]. The very first production bug that is successfully resolved with the help of REPT had been left unfixed for almost two years because developers cannot reproduce the crash locally. The failure occurs in Microsoft Edge when an exception is thrown because a function returns with an error. The bug is hard to fix because there are two possible reasons for the function to fail and it is difficult to tell the actual reason by looking at the memory dump. With the reverse debugging enabled by REPT, the developer is able to step through the function based on the reconstructed execution history and quickly find out the root cause and fix the bug. In summary, a two-year-old bug was fixed in just a few minutes thanks to REPT.

6 Discussion

In this section, we discuss the limitations of REPT and how we plan to address them in future work.

When developers use REPT in practice, they currently have to deal with two main limitations. First, the control flow trace may not be long enough to capture the defect (e.g., the free call is not in the trace for a use-after-free bug). Second, data values that are necessary for debugging the failure are not recovered (e.g., the heap address passed to the free call is not recovered for a use-after-free bug). We cannot simply use a large circular trace buffer to solve this problem because the data recovery accuracy decreases when the trace size increases.

REPT currently does not capture any data during a program's execution. To fundamentally solve these two limitations, we will need to log more data than just the memory dump. It is an open research question to identify a good trade-off between online data logging, runtime overhead, and offline data recovery. A potential direction is to leverage the new `PTWRITE` instruction [36] to log data that is important for REPT's data recovery.

The current implementation of REPT only supports reverse debugging of user-mode executions. While REPT's core analysis is on machine instructions and thus independent of the privilege mode, we need to properly handle kernel-specific artifacts such as interrupts to support reverse debugging of kernel-mode executions.

In addition to reverse debugging, we believe one can leverage the execution history recovered by REPT to perform automatic root cause analysis. The challenge is that the data recovery of REPT is not perfect, so the research question is how to perform automatic root cause analysis based on the imperfect information provided by REPT.

Our evaluation of REPT has been focused on software running on a single machine. When developers debug distributed systems, they usually rely on event logging. It is an interesting research direction to study how program tracing can be combined with event logging to help developers debug bugs in distributed systems. We have not been able to apply REPT to mobile applications because there is no efficient hardware tracing like Intel PT available on mobile devices.

7 Related Work

There is a large body of related work dedicated to debugging failures. More recently, there have been increasing interest in debugging failures in deployed systems. In this section, we discuss some representative examples and describe how REPT differs.

Automatic Root Cause Diagnosis Techniques. A large body of automated root cause diagnosis techniques rely on statistical techniques such as sampling and outlier detection to isolate the key reasons behind a failure and thus help debugging. Cooperative bug isolation [19, 20, 37, 41], failure sketching [40], and lazy diagnosis [39] are state-of-the-art techniques. Unlike these techniques, REPT does not target at a subset of potential bugs or rely on statistical methods to isolate failure causes, but it rather focuses on reconstructing executions. We perceive these techniques as orthogonal and complementary to REPT.

POMP [57] is an automatic root cause analysis tool based on a control flow trace and a memory dump. It handles missing memory writes by running hypothesis tests *recursively*, which significantly limits its efficiency, because the number of hypotheses grows exponentially with the trace size. In contrast, REPT uses a new error correction technique to do forward/backward analysis *iteratively*, which makes its analysis grow linearly with the trace size. We compare their performance on 3 of the 14 bugs (Nasm-2004-1287, PuTTY-2016-2563, and Python-2007-4965) that are evaluated by both. REPT is 1 to 3 orders of magnitude faster than POMP. For instance, POMP takes 30 minutes to analyze the PuTTY-2016-2563 bug, but REPT only takes 5.2 seconds. POMP is evaluated only on how well it works for root cause analysis. There is no instruction-level accuracy reported in the paper, so we cannot directly compare its accuracy with REPT. Furthermore, POMP only supports a single thread, but REPT handles concurrency.

ProRace [62] attempts to recover data values based

on the control flow logged by Intel PT and the register values logged by Intel Processor Event Based Sampling (PEBS) [36]. Unlike REPT, ProRace does not provide solutions for the problems of missing memory writes and concurrent memory writes.

PRES [51] and HOLMES [24] record execution information (e.g., path profiles, function call traces, etc.) to help debug failures. PRES performs state space exploration using the recorded information to reproduce bugs. HOLMES performs bug diagnosis purely based on control flow traces. REPT relies on the lightweight hardware control flow tracing to reconstruct data flows from a memory dump.

"Better Bug Reporting" [23] is a system that performs symbolic execution on a full execution trace to generate a new input that can lead to the same failure. Reporting the generated input instead of the original input can provide better privacy. The main limitation is that it usually introduces high overhead to record a full execution trace. Furthermore, by using a full trace, this bug reporting scheme does not need to handle memory aliasing, but this is not the case for REPT.

Execution Synthesis (ESD) [60] does not assume there is any execution trace. Given a coredump, it relies on heuristics to explore possible paths to search for inputs that may lead to the crash. As recognized in the ESD paper, due to the limitations of symbolic executions for solving complex constraints, ESD may not be able to scale to large programs with long executions.

Delta debugging [61] iteratively isolates program inputs and the control flow of failing executions by repeatedly reproducing the failing and successful runs, and altering variable values. REPT does not make the assumption that failures can be reproduced and operates on a single control flow trace and memory dump.

PSE [42] is a static analysis tool that performs backward slicing and alias analysis on source code to identify potential sources of a NULL pointer. PSE has false positives and is not evaluated on real-world crashes.

Record/Replay Techniques. As we discussed earlier, certain techniques rely on full system record/replay [47–49,56] to help debug failures. REPT does not rely on full system record/replay, which is expensive for deployment usage, but rather reconstructs executions by leveraging lightweight control flow tracing.

Castor [43] is a recent record/replay system that relies on commodity hardware support as well as instrumentation to enable low-overhead recording. Castor works efficiently for programs without data races. In our experience, many programs have data races in practice, which actually make debugging very hard. REPT handles systems with data races.

Ochiai [16] and Tarantula [38] record failing and successful executions and replay them to isolate root causes. REPT does not rely on expensive record/replay techniques nor does it assume bugs can be reproduced.

H3 [35] uses a control flow trace to reduce the constraint complexity for finding a schedule of shared data accesses that can reproduce a failure. H3 does not recover data values, and only applies constraint solving to a small number of shared variables.

State-of-the-Art Techniques in Deployed Systems. Despite extensive prior research, to our knowledge, there are few examples of debugging techniques that are actively used in deployed systems. RETracer [27] is a bug triaging tool that was deployed in Windows Error Reporting [30]. RETracer assigns "blame" to a function for modifying a pointer that ultimately causes an access violation. RETracer performs backward taint analysis based on an approximate execution history recovered by reverse execution. RETracer does not require a control flow trace but can only recover limited data values.

8 Conclusion

We have presented REPT, a practical solution for reverse debugging of software failures in deployed systems. REPT can accurately and efficiently recover data values based on a control flow trace and a memory dump by performing forward and backward execution iteratively with error correction. We implement and deploy REPT into the ecosystem of Microsoft Windows for program tracing, failure reporting, and debugging. Our experiments show that REPT can recover data values with high accuracy in just seconds, and its reverse debugging is effective for diagnosing 14 out of 16 bugs. Given REPT, we hope one day developers will refuse to debug failures without reverse debugging.

9 Acknowledgments

We thank our shepherd, Xi Wang, and other reviewers for their insightful feedback. We are very grateful for all the help from our colleagues on the Microsoft Windows team. In particular, Alan Auerbach, Peter Gilson, Khom Kaowthumrong, Graham McIntyre, Timothy Misiak, Jordi Mola, Prashant Ratanchandani, and Pedro Teixeira provided tremendous help and valuable perspectives throughout the project. We also thank Beeman Strong from Intel for answering numerous questions about Intel Processor Trace.

References

[1] https://bz.apache.org/bugzilla/show_bug.cgi?id=24483.

[2] https://bz.apache.org/bugzilla/show_bug.cgi?id=39722.

[3] https://bz.apache.org/bugzilla/show_bug.cgi?id=60324.

[4] https://www.exploit-db.com/exploits/25005/.

[5] http://ifsec.blogspot.com/2007/04/php-521-wbmp-file-handling-integer.html.

[6] https://www.exploit-db.com/exploits/17201/.

[7] https://bugs.php.net/bug.php?id=74194.

[8] https://bugs.php.net/bug.php?id=76041.

[9] https://github.com/tintinweb/pub/tree/master/pocs/cve-2016-2563.

[10] https://bugs.python.org/issue1179.

[11] https://bugs.python.org/issue28322.

[12] https://bugs.chromium.org/p/chromium/issues/detail?id=784183.

[13] https://bugs.python.org/issue31530.

[14] https://bugs.chromium.org/p/chromium/issues/detail?id=776677.

[15] https://bugs.documentfoundation.org/show_bug.cgi?id=88914.

[16] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim Intl. Symp. on Dependable Computing*, 2006.

[17] Apple Inc. MacOSX CrashReporter. https://developer.apple.com/library/content/technotes/tn2004/tn2123.html, 2017.

[18] Arm Embedded Trace Macrocell (ETM), 2017. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[19] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[20] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7), July 2011.

[22] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[23] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[24] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Intl. Conf. on Software Engineering*, 2009.

[25] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with revgen. In *Proceedings of the 7th Workshop on Hot Topics in System Dependability*, 2011.

[26] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 2010.

[27] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering*, 2016.

[28] J. Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Vienna, Austria, 2012.

[29] J. Gilchrist. Parallel BZIP2. http://compression.ca/pbzip2, 2017.

[30] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *ACM Symp. on Operating Systems Principles*, 2009.

[31] GNU Foundation. GDB and reverse debugging. https://www.gnu.org/software/gdb/news/reversible.html, 2018.

[32] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification*, 2008.

[33] Google Inc. Chrome Error and Crash Reporting. https://support.google.com/chrome/answer/96817?hl=enl, 2017.

[34] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. http://www.cs.wisc.edu/~markhill/racey.html.

[35] S. Huang, B. Cai, and J. Huang. Towards production-run heisenbugs reproduction on commercial hardware. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, 2017. USENIX Association.

[36] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, 2017.

[37] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[38] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[39] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *ACM Symp. on Operating Systems Principles*, Shanghai, China, October 2017.

[40] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *ACM Symp. on Operating Systems Principles*, 2015.

[41] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[42] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM International Symposium on Foundations of Software Engineering*, 2004.

[43] A. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieŕes, and M. Rosenblum. Towards practical default-on multi-core record/replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2017.

[44] Microsoft Corporation. Time travel debugging. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview.

[45] Microsoft Corporation. Windows Debugger. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/.

[46] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Intl. Symp. on Computer Architecture*, 2008.

[47] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[48] Mozilla Corporation. Mozilla rr. http://rr-project.org/, 2017.

[49] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Intl. Symp. on Computer Architecture*, 2005.

[50] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 2009.

[51] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. PRES: Probabilistic replay with execution sketching on multiprocessors. In *ACM Symp. on Operating Systems Principles*, 2009.

[52] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[53] C. Rossi. Rapid release at massive scale. https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/, 2015.

[54] Ubuntu. Ubuntu error. https://wiki.ubuntu.com/ErrorTracker, 2017.

[55] Undo. UndoDB: The interactive reverse debugger for C/C++ on Linux and Android. https://undo.io/, 2018.

[56] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[57] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, 2017. USENIX Association.

[58] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[59] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ACM SIGSOFT European Conference on Foundations of Software Engineering*, 2011.

[60] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM European Conf. on Computer Systems*, 2010.

[61] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[62] T. Zhang, C. Jung, and D. Lee. ProRace: Practical data race detection for production use. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing

Jayashree Mohan* [1] Ashlie Martinez* [1] Soujanya Ponnapalli[1] Pandian Raju[1]
Vijay Chidambaram[1,2]

[1]University of Texas at Austin [2]VMware Research

Abstract

We present a new approach to testing file-system crash consistency: *bounded black-box crash testing* (B^3). B^3 tests the file system in a black-box manner using workloads of file-system operations. Since the space of possible workloads is infinite, B^3 bounds this space based on parameters such as the number of file-system operations or which operations to include, and exhaustively generates workloads within this bounded space. Each workload is tested on the target file system by *simulating* power-loss crashes while the workload is being executed, and checking if the file system recovers to a correct state after each crash. B^3 builds upon insights derived from our study of crash-consistency bugs reported in Linux file systems in the last five years. We observed that most reported bugs can be reproduced using small workloads of three or fewer file-system operations on a newly-created file system, and that all reported bugs result from crashes after `fsync()` related system calls. We build two tools, CRASHMONKEY and ACE, to demonstrate the effectiveness of this approach. Our tools are able to find 24 out of the 26 crash-consistency bugs reported in the last five years. Our tools also revealed 10 *new* crash-consistency bugs in widely-used, mature Linux file systems, seven of which existed in the kernel since 2014. The new bugs result in severe consequences like broken rename atomicity and loss of persisted files.

1 Introduction

A file system is *crash consistent* if it always recovers to a correct state after a crash due to a power loss or a kernel panic. The file-system state is correct if the file system's internal data structures are consistent, and files that were persisted before the crash are not lost or corrupted. When developers added delayed allocation to the ext4 file system [37] in 2009, they introduced a crash-consistency bug that led to wide-spread data loss [24]. Given the potential consequences of crash-consistency bugs and the fact that even professionally-managed datacenters occasionally suffer from power losses [39–42, 60, 61], it is important to ensure that file systems are crash consistent.

Unfortunately, there is little to no crash-consistency testing today for widely-used Linux file systems such as ext4, xfs [55], btrfs [51], and F2FS [25]. The current practice in the Linux file-system community is to not do any *proactive* crash-consistency testing. If a user reports a crash-consistency bug, the file-system developers will then *reactively* write a test to capture that bug. Linux file-system developers use `xfstests` [16], an ad-hoc collection of correctness tests, to perform regression testing. `xfstests` contains a total of 482 correctness tests that are applicable to all POSIX file systems. Of these 482 tests, only 26 (5%) are crash-consistency tests. Thus, file-system developers have no easy way of systematically testing the crash consistency of their file systems.

This paper introduces a new approach to testing file-system crash consistency: *bounded black-box crash testing* (B^3). B^3 is a black-box testing approach: no file-system code is modified. B^3 works by exhaustively generating workloads within a bounded space, *simulating* a crash after persistence operations like `fsync()` in the workload, and finally testing whether the file system recovers correctly from the crash. We implement the B^3 approach by building two tools, CRASHMONKEY and ACE. Our tools are able to find 24 out of the 26 crash-consistency bugs reported in the last five years, across seven kernel versions and three file systems. Furthermore, the systematic nature of B^3 allows our tools to find *new* bugs: CRASHMONKEY and ACE find 10 bugs in widely-used Linux file systems which lead to severe consequences such as `rename()` not being atomic and files disappearing after `fsync()`. We have reported all new bugs; developers have submitted patches for four, and are working to fix the rest.

We formulated B^3 based on our study of all 26 crash-consistency bugs in ext4, xfs, btrfs, and F2FS reported in the last five years (§3). Our study provided key insights

*Both authors contributed equally

that made B^3 feasible: most reported bugs involved a small number of file-system operations on a new file system, with a crash right after a *persistence point* (a call to `fsync()`, `fdatasync()`, or `sync` that flushes data to persistent storage). Most bugs could be found or reproduced simply by systematic testing on a small space of workloads, with crashes only after persistence points. Note that without these insights which bound the workload space, B^3 is infeasible: there are infinite workloads that can be run on infinite file-system images.

Choosing to crash the system only after persistence points is one of the key decisions that makes B^3 tractable. B^3 does not explore bugs that arise due to crashes in the *middle* of a file-system operation because file-system guarantees are undefined in such scenarios. Moreover, B^3 cannot reliably assume that the on-storage file-system state has been modified if there is no persistence point. Crashing only after persistence points bounds the work to be done to test crash consistency, and also provides clear correctness criteria: files and directories which were successfully persisted before the crash must survive the crash and not be corrupted.

B^3 *bounds* the space of workloads in several other ways. First, B^3 restricts the number of file-system operations in the workload, and simulates crashes only after persistence points. Second, B^3 restricts the files and directories that function as arguments to the file-system operations in the workload. Finally, B^3 restricts the initial state of the system to be a small, new file system. Together, these bounds greatly reduce the space of possible workloads, allowing CrashMonkey and Ace to exhaustively generate and test workloads.

An approach like B^3 is only feasible if we can *automatically* and *efficiently* check crash consistency for arbitrary workloads. We built CrashMonkey, a framework that simulates crashes during workload execution and tests for consistency on the recovered file-system image. CrashMonkey first profiles a given workload, capturing all the IO resulting from the workload. It then replays IO requests until a persistence point to create a new file-system image we term a *crash state*. At each persistence point, CrashMonkey also captures a snapshot of files and directories which have been explicitly persisted (and should therefore survive a crash). CrashMonkey then mounts the file system in each crash state, allows the file system to recover, and uses it's own fine-grained checks to validate if persisted data and metadata are available and correct. Thus, CrashMonkey is able to check crash consistency for arbitrary workloads automatically, without any manual effort from the user. This property is key to realizing the B^3 approach.

We built the Automatic Crash Explorer (Ace) to exhaustively generate workloads given user constraints and file-system semantics. Ace first generates a sequence of file-system operations; *e.g.,* a `link()` followed by a `rename()`. Next, Ace fills in the arguments of each file-system operation. It then exhaustively generates workloads where each file-system operation can optionally be followed by an `fsync()`, `fdatasync()`, or a global `sync` command. Finally, Ace adds operations to satisfy any dependencies (*e.g.,* a file must exist before being renamed). Thus, given a set of constraints, Ace generates an exhaustive set of workloads, each of which is tested with CrashMonkey on the target file system.

B^3 offers a new point in the spectrum of techniques addressing file-system crash consistency, alongside verified file systems [8, 9, 53] and model checking [63, 64]. Unlike these approaches, B^3 targets widely deployed file systems written in low-level languages, and does not require annotating or modifying file-system code.

However, B^3 is not without limitations as it is not guaranteed to find all crash-consistency bugs. Currently, Ace's bounds do not expose bugs that require a large number of operations or exhaustion of file-system resources. While CrashMonkey can test such a workload, Ace will not be able to automatically generate the workload. Despite these limitations, we are hopeful that the black-box nature and ease-of-use of our tools will encourage their adoption in the file-system community, unlike model checking and verified file systems. We are encouraged that researchers at Hanyang University are using our tools to test the crash consistency of their research file system, BarrierFS [62].

This paper makes the following contributions:

- A detailed analysis of crash-consistency bugs reported across three widely-used file systems and seven kernel versions in the last five years (§3)
- The bounded black-box crash testing approach (§4)
- The design and implementation of CrashMonkey and Ace[1] (§5)
- Experimental results demonstrating that our tools are able to efficiently find existing and new bugs across widely-used Linux file systems (§6)

2 Background

We first provide some background on file-system crash consistency, why crash-consistency bugs occur, and why it is important to test file-system crash consistency.

Crash consistency. A file system is crash-consistent if a number of invariants about the file-system state hold after a crash due to power loss or a kernel panic [10, 38].

[1] https://github.com/utsaslab/crashmonkey

Typically, these invariants include using resources only after initialization (*e.g.,* path-names point to initialized metadata such as inodes), safely reusing resources after deletion (*e.g.,* two files shouldn't think they both own the same data block), and atomically performing certain operations such as renaming a file. Conventionally, crash consistency is only concerned with internal file-system integrity. A bug that loses previously persisted data would not be considered a crash-consistency bug as long as the file system remains internally consistent. In this paper, we widen the definition to include data loss. Thus, if a file system loses persisted data or files after a crash, we consider it a crash-consistency bug. The Linux file-system developers agree with this wider definition of crash consistency [15, 56]. However, it is important to note that data or metadata that has not been *explicitly persisted* does not fall under our definition; file systems are allowed to lose such data in case of power loss. Finally, there is an important difference between crash-consistency bugs and file-system correctness bugs: crash-consistency bugs *do not* lead to incorrect behavior if no crash occurs.

Why crash-consistency bugs occur. The root of crash consistency bugs is the fact that most file-system operations *only modify in-memory state*. For example, when a user creates a file, the new file exists only in memory until it is explicitly persisted via the `fsync()` call or by a background thread which periodically writes out dirty in-memory data and metadata.

Modern file systems are complex and keep a significant number of metadata-related data structures in memory. For example, btrfs organizes its metadata as B+ trees [51]. Modifications to these data structures are accumulated in memory and written to storage either on `fsync()`, or by a background thread. Developers could make two common types of mistakes while persisting these in-memory structures, which consequently lead to crash-consistency bugs. The first is neglecting to update certain fields of the data structure. For example, btrfs had a bug where the field in the file inode that determined whether it should be persisted was not updated. As a result, `fsync()` on the file became a no-op, causing data loss on a crash [28]. The second is improperly ordering data and metadata when persisting it. For example, when delayed allocation was introduced in ext4, applications that used rename to atomically update files lost data since the rename could be persisted before the file's new data [24]. Despite the fact that the errors that cause crash-consistency bugs are very different in these two cases, the fundamental problem is that some in-memory state that is required to recover correctly is not written to disk.

```
1 create foo
2 link foo bar
3 sync
4 unlink bar
5 create bar
6 fsync bar
7 CRASH!
```

Figure 1: **Example crash-consistency bug**. The figure shows the workload to expose a crash-consistency bug that was reported in the btrfs file system in Feb 2018 [33]. The bug causes the file system to become un-mountable.

POSIX and file-system guarantees. Nominally, Linux file systems implement the POSIX API, providing guarantees as laid out in the POSIX standard [18]. Unfortunately, POSIX is extremely vague. For example, under POSIX it is legal for `fsync()` to *not* make data durable [48]. Mac OSX takes advantage of this legality, and requires users to employ `fcntl(F_FULLFSYNC)` to make data durable [3]. As a result, file systems often offer guarantees above and beyond what is required by POSIX. For example, on ext4, persisting a new file will also persist its directory entry. Unfortunately, these guarantees vary across different file systems, so we contacted the developers of each file system to ensure we are testing the guarantees that they seek to provide.

Example of a crash-consistency bug. Figure 1 shows a crash-consistency bug in btrfs that causes the file system to become un-mountable (unavailable) after the crash. Resolving the bug requires file-system repair using `btrfs-check`; for lay users, this requires guidance of the developers [7]. This bug occurs on btrfs because the unlink affects two different data structures which become out of sync if there is a crash. On recovery, btrfs tries to unlink `bar` twice, producing an error.

Why testing crash consistency is important. File-system researchers are developing new crash-consistency techniques [13, 14, 46] and designing new file systems that increase performance [1, 5, 21, 23, 50, 54, 68, 69]. Meanwhile, Linux file systems such as btrfs include a number of optimizations that affect the ordering of IO requests, and hence, crash consistency. However, crash consistency is subtle and hard to get right, and a mistake could lead to silent data corruption and data loss. Thus, changes affecting crash consistency should be carefully tested.

State of crash-consistency testing today. `xfstests` [16] is a regression test suite to check file-system correctness, with a small proportion (5%) of crash-consistency tests. These tests are aimed at avoiding the recurrence of the same bug over time, but

Consequence	# bugs
Corruption	19
Data Inconsistency	6
Un-mountable file system	3
Total	28

Kernel Version	# bugs
3.12	3
3.13	9
3.16	1
4.1.1	2
4.4	9
4.15	3
4.16 (latest)	1
Total	28

File System	# bugs
ext4	2
F2FS	2
btrfs	24
Total	28

# of ops required	# bugs
1	3
2	14
3	9
Total	26

Table 1: **Analyzing crash-consistency bugs**. The tables break down the 26 unique crash-consistency bugs reported over the last five years (since 2013) by different criteria. Two bugs were reported on two different file systems, leading to a total of 28 bugs.

do not generalize to identifying variants of the bug. Additionally, each of these test cases requires the developer to write a checker describing the correct behavior of the file system after a crash. Given the infinite space of workloads, it is extremely hard to handcraft workloads that could reveal bugs. These factors make `xfstests` insufficient to identify *new* crash-consistency bugs.

3 Studying Crash-Consistency Bugs

We present an analysis of 26 unique crash-consistency bugs reported by users over the last five years on widely-used Linux file systems [58]. We find these bugs either by examining mailing list messages or looking at the crash-consistency tests in the `xfstests` regression test suite. Few of the crash-consistency tests in `xfstests` link to the bugs that resulted in the test being written.

Due to the nature of crash-consistency bugs (all in-memory information is lost upon crash), it is hard to tie them to a specific workload. As a result, the number of reported bugs is low. We believe there are many crash-consistency bugs that go unreported in the wild.

We analyze the bugs based on consequence, kernel version, file system, and the number of file-system operations required to reproduce them. There are 26 unique bugs spread across ext4, F2FS, and btrfs. Each unique bug requires a unique set of file-system operations to reproduce. Two bugs occur on two file systems (F2FS and ext4, F2FS and btrfs), leading to a total of 28 bugs.

Table 1 presents some statistics about the crash-consistency bugs. The table presents the kernel version in which the bug was reported. If the bug report did not include a version, it presents the latest kernel version in which B^3 could reproduce the bug (the two bugs that B^3 could not reproduce appear in kernel 3.13). The bugs have severe consequences, ranging from file-system corruption to the file system becoming un-mountable. The four most common file-system operations involved in crash-consistency bugs were `write()`, `link()`, `unlink()`, and `rename()`. Most reported bugs resulted from either reusing filenames in multiple file-system operations or write operations to overlapping file regions. Most reported bugs could be reproduced with three or fewer file-system operations.

Examples. Table 2 showcases a few of the crash-consistency bugs. Bug #1 [27] involves creating two files in a directory and persisting only one of them. btrfs log recovery incorrectly counts the directory size, making the directory un-removable thereafter. Bug #2 [29] involves creating a hard link to an already existing file. A crash results in btrfs recovering the file with a size 0, thereby making its data inaccessible. A similar bug (#5 [19]) manifests in ext4 in the direct write path, where the write succeeds and blocks are allocated, but the file size is incorrectly updated to be zero, leading to data loss.

Complexity leads to bugs. The ext4 file system has undergone more than 15 years of development, and, as a result, has only two bugs. The btrfs and F2FS file systems are more recent: btrfs was introduced in 2007, while F2FS was introduced in 2012. In particular, btrfs is an extremely complex file system that provides features such as snapshots, cloning, out-of-band deduplication, and compression. btrfs maintains its metadata (such as inodes and bitmaps) in the form of various copy-on-write B+ trees. This makes achieving crash consistency tricky, as the updates have to be propagated to several trees. Thus, it is not surprising that most reported crash-consistency bugs occurred in btrfs. As file systems become more complex in the future, we expect to see a corresponding increase in crash-consistency bugs.

Crash-consistency bugs are hard to find. Despite the fact that the file systems we examined were widely used, some bugs have remained hidden in them for years. For example, btrfs had a crash-consistency bug that was only discovered *seven* years after it was introduced. The bug was caused by incorrectly processing a hard link in

Bug #	File System	Consequence	# of ops	ops involved (excluding persistence operations)
1	btrfs	Directory un-removable	2	`creat(A/x), creat(A/y)`
2	btrfs	Persisted data lost	2	`pwrite(x), link(x,y)`
3	btrfs	Directory un-removable	3	`link(x,A/x), link(x,A/y), unlink(A/y)`
4	F2FS	Persisted file disappears	3	`pwrite(x), rename(x,y), pwrite(x)`
5	ext4	Persisted data lost	2	`pwrite(x), direct_write(x)`

Table 2: **Examples of crash-consistency bugs**. The table shows some of the crash-consistency bugs reported in the last five years. The bugs have severe consequences, ranging from losing user data to making directories un-removable.

btrfs's data structures. When a hard link is added, the directory entry is added to one data structure, while the inode is added to another data structure. When a crash occurred, only one of these data structures would be correctly recovered, resulting in the directory containing the hard link becoming un-removable [30]. This bug was present since the log tree was added in 2008; however, the bug was only discovered in 2015.

Systematic testing is required. Once the hard link bug in btrfs was discovered, the btrfs developers quickly fixed it. However, they only fixed one code path that could lead to the bug. The same bug could be triggered in another code path, a fact that was only discovered *four months* after the original bug was reported. While the original bug workload required creating hard links and calling `fsync()` on the original file and parent directory, this one required calling `fsync()` on a sibling in the directory where the hard link was created [31]. Systematic testing of the file system would have revealed that the bug could be triggered via an alternate code path.

Small workloads can reveal bugs on an empty file system. Most of the reported bugs do not require a special file-system image or a large number of file-system operations to reproduce. 24 out of the 26 reported bugs require three or fewer core file-system operations to reproduce on an empty file system. This count is low because we do not count *dependent* operations: for example, a file has to exist before being renamed and a directory has to exist before a file can be created inside it. Such dependent operations can be *inferred* given the core file-system operations. Of the remaining two bugs, one required a special command (`dropcaches`) to be run during the workload for the bug to manifest. The other bug required specific setup: 3000 hard links had to already exist (forcing an external reflink) for the bug to manifest.

Reported bugs involve a crash after persistence. All reported bugs involved a crash right after a persistence point: a call to `fsync()`, `fdatasync()`, or the global `sync` command. These commands are important because file-system operations only modify in-memory metadata and data by default. Only persistence points reliably change the file-system state on storage. Therefore, unless a file or directory has been persisted, it cannot be expected to survive a crash. While crashes could technically occur at any point, a user cannot complain if a file that has not been persisted goes missing after a crash. Thus, every crash-consistency bug involves *persisted data or metadata* that is affected by the bug after a crash, and a workload that does not have a persistence point cannot lead to a reproducible crash-consistency bug. This also points to an effective way to find crash-consistency bugs: perform a sequence of file-system operations, change on-storage file-system state with `fsync()` or similar calls, crash, and then check files and directories that were previously persisted.

4 B^3: Bounded Black-Box Crash Testing

Based on the insights from our study of crash-consistency bugs, we introduce a new approach to testing file-system crash consistency: *Bounded Black-Box crash testing* (B^3). B^3 is a black-box testing approach built upon the insight that most reported crash-consistency bugs can be found by systematically testing small sequences of file-system operations on a new file system. B^3 exercises the file system through its system-call API, and observes the file-system behavior via read and write IO. As a result, B^3 does not require annotating or modifying file-system source code.

4.1 Overview

B^3 generates sequences of file-system operations, called *workloads*. Since the space of possible workloads is infinite, B^3 *bounds* the space of workloads using insights from the study. Within the determined bounds, B^3 exhaustively generates and tests all possible workloads. Each workload is tested by *simulating* a crash after each persistence point, and checking if the file system recovers to a correct state. B^3 performs fine-grained correctness checks on the recovered file-system state; only files and

directories that were explicitly persisted are checked. B^3 checks for both data and metadata (size, link count, and block count) consistency for files and directories.

Crash points. The main insight from the study that makes an approach like B^3 feasible is the choice of crash points; a crash is simulated *only* after each persistence point in the workload instead of in the middle of file-system operations. This design choice was motivated by two factors. First, file-system guarantees are undefined if a crash occurs in the middle of a file-system operation; only files and directories that were previously successfully persisted need to survive the crash. File-system developers are overloaded, and bugs involving data or metadata that has not been explicitly persisted is given low priority (and sometimes not acknowledged as a bug). Second, if we crash in the middle of an operation, there are a number of correct states the file system could recover to. If a file-system operation translates to n block IO requests, there could be 2^n different on-disk crash states if we crashed anywhere during the operation. Restricting crashes to occur after persistence points bounds this space linearly in the number of operations comprising the workload. The small set of crash points and correct states makes automated testing easier. Our choice of crash points naturally leads to bugs where persisted data and metadata is corrupted or missing and file-system developers are strongly motivated to fix such bugs.

4.2 Bounds used by B^3

Based on our study of crash-consistency bugs, B^3 bounds the space of possible workloads in several ways:

1. **Number of operations**. B^3 bounds the number of file-system operations (termed the *sequence length*) in the workload. A `seq-X` workload has X core file-system operations in it, not counting dependent operations such as creating a file before renaming it.
2. **Files and directories in workload**. We observe that in the reported bugs, errors result from the *reuse* of a small set of files for metadata operations. Thus, B^3 restricts workloads to use few files per directory, and a low directory depth. This restriction automatically reduces the inputs for metadata-related operations such as `rename()`.
3. **Data operations**. The study also indicated that bugs related to data inconsistency mainly occur due to writes to *overlapping* file ranges. In most cases, the bugs are not dependent on the exact offset and length used in the writes, but on the interaction between the overlapping regions from writes. The study indicates that a broad classification of writes such as appends to the end of a file, overwrites to overlapping regions of file, *etc.* is sufficient to find crash-consistency bugs.
4. **Initial file-system state**. Most of the bugs analyzed in the study did not require a specific initial file-system state (or a large file system) to be revealed. Moreover, most of the studied bugs could be reproduced starting from the *same, small* file-system image. Therefore, B^3 can test all workloads starting from the same initial file-system state.

4.3 Fine-grained correctness checking

B^3 uses fine-grained correctness checks to validate the data and metadata of persisted files and directories in each crash state. Since `fsck` is both time-consuming to run and can miss data loss/corruption bugs, it is not a suitable checker for B^3.

4.4 Limitations

The B^3 approach has a number of limitations:

1. B^3 does not make any guarantees about finding *all* crash-consistency bugs. It is sound but incomplete. However, because B^3 tests exhaustively, if the workload that triggers the bug falls within the constrained workload space, B^3 will find it. Therefore, the effectiveness of B^3 depends upon the bounds chosen and the number of workloads tested.
2. B^3 focuses on a specific class of bugs. It does not simulate a crash in the middle of a file-system operation and it does not re-order IO requests to create different crash states. The implicit assumption is that the core crash-consistency mechanism, such as journaling [49] or copy-on-write [20, 52], is working correctly. Instead, we assume that it is the rest of the file system that has bugs. The crash-consistency bug study indicates this assumption is reasonable.
3. B^3 focuses on workloads where files and directories are explicitly persisted. If we created a file, waited one hour, then crashed, and found that the file was gone after the file-system recovered, this would also be a crash-consistency bug. However, B^3 does not explore such workloads as they take a significant amount of time to run and are not easily reproduced in a deterministic fashion.
4. Due to its black-box nature, B^3 cannot pinpoint the exact lines of code that result in the observed bug. Once a bug has been revealed by B^3, finding the root cause requires further investigation. However, B^3 aids in investigating the root cause of the bug since it provides a way to reproduce the bug in a deterministic fashion.

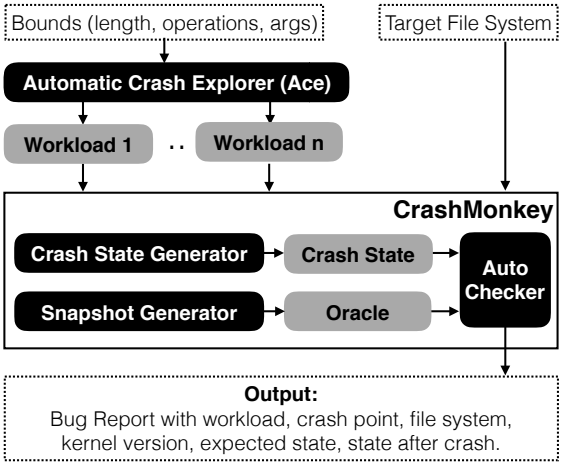


Figure 2: **System architecture**. Given bounds for exploration, ACE generates a set of workloads. Each workload is then fed to CRASHMONKEY, which generates a set of crash states and corresponding oracles. The AutoChecker compares persisted files in each oracle/crash state pair; a mismatch indicates a bug.

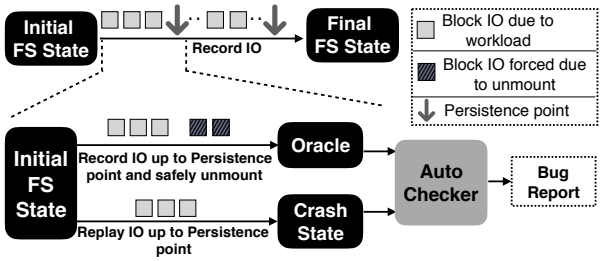


Figure 3: **CRASHMONKEY operation**. CRASHMONKEY first records the block IO requests that the workload translates to, capturing reference images called oracles after each persistence point. CRASHMONKEY then generates crash states by replaying the recorded IO and tests for consistency against the corresponding oracle.

Despite its shortcomings, we believe B^3 is a useful addition to the arsenal of techniques for testing file-system crash consistency. The true strengths of B^3 lie in its systematic nature and the fact that it does not require any changes to existing systems. Therefore, it is ideal for complex and widely-used file systems written in low-level languages like C, where stronger approaches like verification cannot be easily used.

5 CrashMonkey and Ace

We realize the B^3 approach by building two tools, CRASHMONKEY and ACE. As shown in Figure 2, CRASHMONKEY is responsible for simulating crashes at different points of a given workload and testing if the file system recovers correctly after each simulated crash, while the Automatic Crash Explorer (ACE) is responsible for exhaustively generating workloads in a bounded space.

5.1 CrashMonkey

CRASHMONKEY uses record-and-replay techniques to *simulate* a crash in the middle of the workload and test if the file system recovers to a correct state after the crash. For maximum portability, CRASHMONKEY treats the file system as a black box, only requiring that the file system implement the POSIX API.

Overview. CRASHMONKEY operates in three phases as shown in Figure 3. In the first phase, CRASHMONKEY profiles the workload by collecting information about all file-system operations and IO requests made during the workload. The second phase replays IO requests until a persistence point to create a *crash state*. The crash state represents the state of storage if the system had crashed after a persistence operation completed. CRASHMONKEY then mounts the file system in the crash state and allows the file system to perform recovery. At each persistence point, CRASHMONKEY also captures a reference file-system image, termed the *oracle*, by safely unmounting it so the file system completes any pending operations or checkpointing. The oracle represents the expected state of the file system after a crash. In the absence of bugs, persisted files should be the same in the oracle and the crash state after recovery. In the third phase, CRASHMONKEY's AutoChecker tests for correctness by comparing the persisted files and directories in the oracle with the crash state after recovery.

CRASHMONKEY is implemented as two kernel modules and a set of user-space utilities. The kernel modules consist of 1300 lines of C code which can be compiled and inserted into the kernel at run time, thus avoiding the need for long kernel re-compilations. The user-space utilities consist of 4800 lines of C++ code. CRASHMONKEY's separation into kernel modules and user-space utilities allows rapid porting to a different kernel version; only the kernel modules need to be ported to the target kernel. This allowed us to port CRASHMONKEY to seven kernels to reproduce the bugs studied in §3.

Profiling workloads. CRASHMONKEY profiles workloads at two levels of the storage stack: it records block IO requests, and it records system calls. It uses two kernel modules to record block IO requests and create crash states and oracles.

The first kernel module records all IO requests generated by the workload using a wrapper block device on

B^3 bound	Insight from the study	Bound chosen by ACE
Number of operations	Small workloads of 2-3 core operations	Maximum # of core ops in a workload is *three*
Files and directories	Reuse file and directory names	2 directories of depth 2, each with 2 unique files
Data operations	Coarse grained, overlapping ranges of writes	Overwrites to start, middle & end of file, and appends
Initial file-system state	No need of a special initial state or large image	Start with a clean file-system image of size 100MB

Table 3: **Bounds used by ACE**. The table shows the specific values picked by ACE for each B^3 bound.

which the target file system is mounted. The wrapper device records both data and metadata for IO requests (such as sector number, IO size, and flags). Each persistence point in the workload causes a special *checkpoint* request to be inserted into the stream of IO requests recorded. The checkpoint is simply an empty block IO request with a special flag, to correlate the completion of a persistence operation with the low-level block IO stream. All the data recorded by the wrapper device is communicated to the user-space utilities via `ioctl` calls.

The second kernel module in CRASHMONKEY is an in-memory, copy-on-write block device that facilitates snapshots. CRASHMONKEY creates a snapshot of the file system before the profiling phase begins, which represents the base disk image. CRASHMONKEY provides fast, writable snapshots by replaying the IO recorded during profiling on top of the base disk image to generate a crash state. Snapshots are also saved at each persistence point in the workload to create oracles. Furthermore, since the snapshots are copy-on-write, resetting a snapshot to the base image simply means dropping the modified data blocks, making it efficient.

CRASHMONKEY also records all `open()`, `close()`, `fsync()`, `fdatasync()`, `rename()`, `sync()`, and `msync()` calls in the workload so that when the workload does a persistence operation such as `fsync(fd)`, CRASHMONKEY is able to correlate `fd` with a file that was opened earlier. This allows CRASHMONKEY to track the set of files and directories that were explicitly persisted at any point in the workload. This information is used by CRASHMONKEY's AutoChecker to ensure that only files and directories explicitly persisted at a given point in the workload are compared. CRASHMONKEY uses its own set of functions that wrap system calls which manipulate files to record the required information.

Constructing crash states. To create a crash state, CRASHMONKEY starts from the initial state of the file system (before the workload was run), and uses a utility similar to `dd` to replay all recorded IO requests from the start of the workload until the next checkpoint in the IO stream. The resultant crash state represents the state of the storage just after the persistence-related call completed on the storage device. Since the IO stream replay ends directly after the next persistence point in the stream, the generated crash point represents a file-system state that is considered uncleanly unmounted. Therefore, when the file system is mounted again, the kernel may run file-system specific recovery code.

Automatically testing correctness. CRASHMONKEY's AutoChecker is able to test for correctness automatically because it has three key pieces of information: it knows which files were persisted, it has the correct data and metadata of those files in the oracle, and it has the actual data and metadata of the corresponding files in the crash state after recovery. Testing correctness is a simple matter of comparing data and metadata of persisted files in the oracle and the crash state.

CRASHMONKEY avoids using `fsck` because its runtime is proportional to the amount of data in the file system (not the amount of data changed) and it does not detect the loss or corruption of user data. Instead, when a crash state is re-mounted, CRASHMONKEY allows the file system to run its recovery mechanism, like journal replay, which is usually more lightweight than `fsck`. `fsck` is run only if the recovered file system is unmountable. To check consistency, CRASHMONKEY uses its own *read* and *write* checks after recovery. The read checks used by CRASHMONKEY confirm that persisted files and directories are accurately recovered. The write checks test if a bug makes it impossible to modify files or directories. For example, a btrfs bug made a directory un-removable due to a stale file handle [27].

Since each file system has slightly different consistency guarantees, we reached out to developers of each file system we tested, to understand the guarantees provided by that file system. In some cases, our conversations prompted the developers to explicitly write down the persistence guarantees of their file systems for the first time [57]. During this process, we confirmed that most file systems such as ext4 and btrfs implement a stronger set of guarantees than the POSIX standard. For example, while POSIX requires an `fsync()` on both a newly created file and its parent directory to ensure the file is present after a crash, many Linux file systems do

not require the `fsync()` of the parent directory. Based on the response from developers, we report bugs that violate the guarantees each file system aims to provide.

5.2 Automatic Crash Explorer (Ace)

ACE exhaustively generates workloads satisfying the given bounds. ACE has two components, the workload synthesizer and the adapter for CRASHMONKEY.

Workload synthesizer. The workload synthesizer exhaustively generates workloads within the state space defined by the user specified bounds. The workloads generated in this stage are represented in a high-level language, similar to the one depicted in Figure 4.

CrashMonkey Adapter. A custom adapter converts the workload generated by the synthesizer into an equivalent C++ test file that CRASHMONKEY can work with. This adapter handles the insertion of wrapped file-system operations that CRASHMONKEY tracks. Additionally, it inserts a special function-call at every persistence point, which translates to the checkpoint IO. It is easy to extend ACE to be used with other record-and-replay tools like `dm-log-writes` [4] by building custom adapters.

Table 3 shows how we used the insights from the study to assign specific values for B^3 bounds when we run ACE. Given these bounds, ACE uses a multi-phase process to generate workloads that are then fed into CRASHMONKEY. Figure 4 illustrates the four phases ACE goes through to generate a `seq-2` workload.

Phase 1: Select operations and generate workloads. ACE first selects file-system operations for the given sequence length to make what we term the *skeleton*. By default, file-system operations can be repeated in the workload. The user may also supply bounds such as requiring only a subset of file-system operations be used (*e.g.,* to focus testing on new operations). ACE then exhaustively generates workloads satisfying the given bounds. For example, if the user specified the `seq-2` workload could only contain six file-system operations, ACE will generate $6 * 6 = 36$ skeletons in phase one.

Phase 2: Select parameters. For each skeleton generated in phase one, ACE then selects the parameters (system-call arguments) for each file-system operation. By default, ACE uses two files at the top level and two sub-directories with two files each as arguments for metadata-related operations. ACE also understands the semantics of file-system operations and exploits it to eliminate the generation of *symmetrical* workloads. For example, consider two operations `link(foo, bar)` and `link(bar, foo)`. The idea is to link two files within the same directory, but the order of file names chosen does not matter. In this example, one of the workloads would be discarded, thus reducing the total number of workloads to be tested for the sequence.

For data operations, ACE chooses between whether a write is an overwrite at the beginning, middle, or end of the file or simply an append operation. Furthermore, since our study showed that crash-consistency bugs occur when data operations overlap, ACE tries to overlap data operations in phase two.

Each skeleton generated in phase one can lead to multiple workloads (based on different parameters) in phase two. However, at the end of this phase, each generated workload has a sequence of file-system operations with all arguments identified.

Phase 3: Add persistence points. ACE optionally adds a persistence point after each file-system operation in the workload, but ACE does not require every operation to be followed by a persistence point. However, ACE ensures that the last operation in a workload is always followed by a persistence point so that it is not truncated to a workload of lower sequence length. The file or directory to be persisted in each call is selected from the same set of files and directories used by phase two, and, for each workload generated by phase two, phase three can generate multiple workloads by adding persistence points after different sets of file-system operations.

Phase 4: Add dependencies. Finally, ACE satisfies various dependencies to ensure the workload can execute on a POSIX file system. For example, a file has to exist before being renamed or written to. Similarly, directories have to be created if any operations on their files are involved. Figure 4 shows how `A`, `B`, and `A/foo` are created as dependencies in the workload. As a result, a `seq-2` workload can have more than two file-system operations in the final workloads. At the end of this phase, ACE compiles each workload from the high-level language into a C++ program that can be passed to CRASHMONKEY.

Implementation. ACE consists of 2500 lines of Python code, and currently supports 14 file-system operations. All bugs analyzed in our study used one of these 14 file-system operations. It is straightforward to expand ACE to support more operations.

Running Ace with relaxed bounds. It is easy to relax the bounds used by ACE to generate more workloads; this comes at the cost of computational time used to test the extra workloads. Care should be taken when relaxing the bounds, since the number of workloads increases at a rapid rate. For example, ACE generates about 1.5M workloads with three core file-system operations. Relaxing the default bound on the set of files and direc-

Phase 1: Select operations	**Phase 2:** Select parameters	**Phase 3:** Add persistence points	**Phase 4:** Add dependencies

`1 rename()`
`2 link()`
	`1 rename(A/foo,B/bar)`
	`2 link(B/bar, A/bar)`
		`1 rename(A/foo,B/bar)`
		`sync()`
		`2 link(B/bar, A/bar)`
		`fsync(A/bar)`
			`mkdir(A)`
			`mkdir(B)`
			`create(A/foo)`
			`1 rename(A/foo,B/bar)`
			`sync()`
			`2 link(B/bar, A/bar)`
			`fsync(A/bar)`

Figure 4: **Workload generation in ACE.** The figure shows the different phases involved in workload generation in ACE. Given the sequence length, ACE first selects the operations, then selects the parameters for each operation, then optionally adds persistence points after each operation, and finally satisfies file and directory dependencies for the workload. The final workload may have more operations than the original sequence length.

tories to add one additional nested directory, increases the number of workloads generated to 3.7M. This simple change results in 2.5× more workloads. Note that increasing the number file-system operations in the workload leads to an increase in the number of phase-1 skeletons generated, and adding more files to the argument set increase the number of phase-2 workloads that can be created. Therefore, the workload space must be carefully expanded.

5.3 Testing and Bug Analysis

Testing Strategy. Given a target file system, we first exhaustively generate `seq-1` workloads and test them using CRASHMONKEY. We then proceed to `seq-2`, and then `seq-3` workloads. By generating and testing workloads in this order, CRASHMONKEY only needs to simulate a crash at one point per workload. For example, even if a `seq-2` workload has two persistence points, crashing after the first persistence point would be equivalent to an already-explored `seq-1` workload.

Analyzing Bug Reports. One of the challenges with a black-box approach like B^3 is that a single bug could result in many different workloads failing correctness tests. We present two cases of multiple test failures in workloads, and how we mitigate them.

First, workloads in different sequences can fail because of the same bug. Our testing strategy is designed to mitigate this: if a bug causes incorrect behavior with a single file-system operation, it should be caught by a `seq-1` workload. Therefore, if we catch a bug only in a `seq-2` workload, it implies the bug results from the interaction of the two file-system operations. Ideally, we would run `seq-1`, report any bugs, and apply bug-fix patches given by developers before running `seq-2`. However, for quicker testing, ACE maintains a database of all previously found bugs which includes the core file-system operations that produced each bug and the consequence of the bug. For all new bugs reports generated by CRASHMONKEY and ACE, it first compares the workload and the consequence with the database of known bugs. If there is a match, ACE does not report the bug to the user.

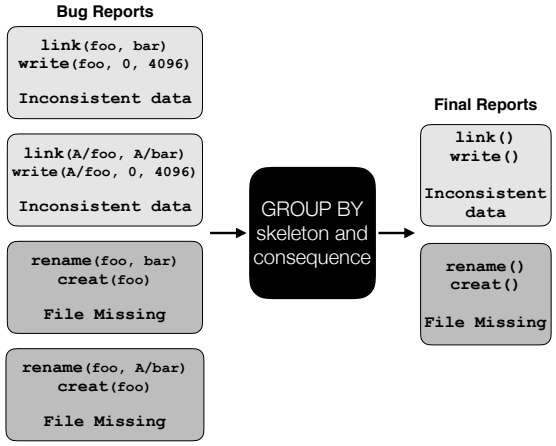


Figure 5: **Post-processing**. The figure shows how generated bug reports are processed to eliminate duplicates.

Second, similar workloads in the same sequence could fail correctness tests due to the same bug. For efficient analysis, we group together bug reports by the consequence (*e.g.,* file missing), and the skeleton (the sequence of core file-system operations that comprise the workload) that triggered the bug, as shown in Figure 5. Using the skeleton instead of the fully fleshed-out workload allows us to identify similar bugs. For example, the bug that causes appended data to be lost will repeat four times, once with each of the files in our file set. We can group these bug reports together and only inspect one bug report from each group. After verifying each bug, we report it to developers.

6 Evaluation

We evaluate the utility and performance of the B^3 approach by answering the following questions:

- Do CrashMonkey and Ace find known bugs and new bugs in Linux file systems in a reasonable period of time? (§6.2)
- What is the performance of CrashMonkey? (§6.3)
- What is the performance of Ace? (§6.4)
- How much memory and CPU does CrashMonkey consume? (§6.5)

6.1 Experimental Setup

B^3 requires testing a large number of workloads in a systematic manner. To accomplish this testing, we deploy CrashMonkey on Chameleon Cloud [26], an experimental testbed for large-scale computation.

We employ a cluster of 65 nodes on Chameleon Cloud. Each node has 40 cores, 48 GB RAM, and 128 GB SSD. We install 12 VirtualBox virtual machines running Ubuntu 16.04 LTS on each node, each with 2 GB RAM and 10 GB storage. Each virtual machine runs one instance of CrashMonkey. Thus, we have a total of 780 virtual machines testing workloads with CrashMonkey in parallel. We found we are limited to 780 virtual machines by the storage available to each physical node.

On a local server, we generate the workloads with Ace and divide them into sets of workloads to be tested on each virtual machine. We then copy the workloads over the network to each physical Chameleon node, and, from each node, copy them to the virtual machines.

6.2 Bug Finding

Determining Workloads. Our goal was to test whether the B^3 approach was useful and practical, not to exhaustively find every crash-consistency bug. Therefore, we wanted to limit the computational time spent on testing to a few days. Thus, we needed to determine what workloads to test with our computational budget.

Our study of crash-consistency bugs indicated that it would be useful to test small workloads of length one, two, and three. However, we estimated that testing all 25 million possible workloads of length three was infeasible within our target time-frame. We had to further restrict the set of workloads that we tested. We used our study to guide us in this task. At a minimum, we wanted to select bounds that would generate the workloads that reproduced the reported bugs. Using this as a guideline, we came up with a set of workloads that was broad enough to reproduce existing bugs (and potentially find new bugs), but small enough that we could test the workloads in a few days on our research cluster.

Workloads. We test workloads of length one (`seq-1`), two (`seq-2`), and three (`seq-3`). We further separate workloads of length three into three groups: one focusing on data operations (`seq-3-data`), one focusing on metadata operations (`seq-3-metadata`), and one focusing on metadata operations involving a file at depth three (`seq-3-nested`) (by default, we use depth two).

The `seq-1` and `seq-2` workloads use a set of 14 file-system operations. For `seq-3` workloads, we narrow down the list of operations, based on what category the workload is in. The complete list of file-system operations tested in each category is shown in Table 4.

Testing Strategy. We tested `seq-1` and `seq-2` workloads on ext4, xfs, F2FS, and btrfs, but did not find any new bugs in ext4 or xfs. We focused on F2FS and btrfs for the larger `seq-3` workloads. In total, we spend 48 hours testing all 3.37 million workloads per file system on the 65-node research cluster described earlier. Table 4 presents the number of workloads in each set, and the time taken to test them (for each file system). All the tests are run only on 4.16 kernel. To reproduce reported bugs, we employ the following strategy. We encode the workload that triggers previously reported bugs in Ace. In the course of workload generation, when Ace generates a workload identical to the encoded one, it is added to a list. This list of workloads is run on the kernel versions reported in Table 1, to validate that the workload produced by Ace can indeed reproduce the bug.

Cost of Computation. We believe the amount of computational effort required to find crash-consistency bugs with CrashMonkey and Ace is reasonable. For example, if we were to rent 780 `t2.small` instances on Amazon to run Ace and CrashMonkey for 48 hours, at the current rate of \$0.023 per hour for on-demand instances [2], it would cost $780 * 48 * 0.023 = \$861.12$. For the complete 25M workload set, the cost of computation would go up by 7.5×, totaling \$6.4K. Thus, we can test each file system for less than \$7K. Alternatively, a company can provision physical nodes to run the tests; we believe this would not be hard for a large company.

Results. CrashMonkey and Ace found 10 **new** crash-consistency bugs [59] in btrfs and F2FS, in addition to reproducing 24 out of 26 bugs reported over the past five years. We studied the bug reports for the new bugs to ensure they were unique and not different manifestations of the same underlying bug. We verified each unique bug triggers a different code path in the kernel, indicating the root cause of each bug is not the same underlying code.

All new bugs were reported to file-system developers and acknowledged [11, 12, 43, 44]. Developers have

Sequence type	File-system operations tested	# of workloads	Run time (minutes)
seq-1	creat, mkdir, falloc, buffered write, mmap, link	300	1
seq-2	direct-IO write, unlink, rmdir, setxattr removexattr, remove, unlink, truncate	254K	215
seq-3-data	buffered write, mmap, direct-IO write, falloc	120K	102
seq-3-metadata	buffered write, link, unlink, rename	1.5M	1274
seq-3-nested	link, rename	1.5M	1274
Total		3.37M	2866

Table 4: **Workloads tested**. The table shows the number of workloads tested in each set, along with the time taken to test these workloads in parallel on 65 physical machines and the file-system operations tested in each category. Overall, we tested 3.37 million workloads in two days, reproducing 24 known bugs and finding 10 new crash-consistency bugs.

submitted patches for four bugs [32, 35, 66, 67], and are working on patches for the others [34]. Table 5 presents the new bugs discovered by CrashMonkey and Ace. We make several observations based on these results.

The discovered bugs have severe consequences. The newly discovered bugs result in either data loss (due to missing files or directories) or file-system corruption. More importantly, the missing files and directories have been *explicitly persisted* with an `fsync()` call and thus should survive crashes.

Small workloads are sufficient to reveal new bugs. One might expect only workloads with two or more file-system operations to expose bugs. However, the results show that even workloads consisting of a single file-system operation, if tested systematically, can reveal bugs. For example, three bugs were found by `seq-1` workloads, where CrashMonkey and Ace only tested 300 workloads in a systematic fashion. Interestingly, variants of these bugs have been patched previously, and it was sufficient to simply change parameters to file-system operations to trigger the same bug through a different code-path.

An F2FS bug found by CrashMonkey and Ace is a good example of finding variants of previously patched bugs. The previously patched bug manifested when `fallocate()` was used with the `KEEP_SIZE` flag; this allocates blocks to a file but does not increase the file size. By calling `fallocate()` with the `KEEP_SIZE` flag, developers found that F2FS only checked the file size to see if a file had been updated. Thus, `fdatasync()` on the file would have no result. After a crash, the file recovered to an incorrect size, thereby not respecting the `KEEP_SIZE` flag. This bug was patched in Nov 2017 [65]; however, the `fallocate()` system call has several more flags like `ZERO_RANGE`, `PUNCH_HOLE`, *etc.*, and developers failed to systematically test all possible parameter options of the system call. Therefore, our tools identified and reported that the same bug can appear when `ZERO_RANGE` is used. Though this bug was recently patched by developers, it provides more evidence that the state of crash-consistency testing today is insufficient, and that systematic testing is required.

Crash-consistency bugs are hard to find manually. CrashMonkey and Ace found eight new bugs in btrfs in kernel 4.16. Interestingly, seven of these bugs have been present since kernel 3.13, which was released in 2014. The ability of our tools to find *four-year-old* crash-consistency bugs within two days of testing on a research cluster of modest size speaks to both the difficulty of manually finding these bugs, and the power of systematic approaches like B^3.

Broken rename atomicity bug. Ace generated several workloads that broke the rename atomicity of btrfs. The workloads consist of first creating and persisting a file such as `A/bar`. Next, the workload creates another file `B/bar`, and tries to replace the original file, `A/bar`, with the new file. The expectation is that we are able to read either the original file, `A/bar`, or the new file, `B/bar`. However, btrfs can lose both `A/bar` and `B/bar` if it crashes at the wrong time. While losing rename atomicity is bad, the most interesting part of this bug is that `fsync()` must be called on an un-related sibling file, like `A/foo`, before the crash. This shows that workloads revealing crash-consistency bugs are hard for a developer to find manually since they don't always involve obvious sequences of operations.

Bug #	File System	Consequence	# of ops	Bug present since
1	btrfs	Rename atomicity broken (file disappears)	3	2014
2	btrfs	Rename atomicity broken (file in both locations)	3	2018
3	btrfs	Directory not persisted by fsync*	3	2014
4	btrfs	Rename not persisted by fsync	3	2014
5	btrfs	Hard links not persisted by fsync	2	2014
6	btrfs	Directory entry missing after fsync on directory	2	2014
7	btrfs	Fsync on file does not persist all its paths	1	2014
8	btrfs	Allocated blocks lost after fsync*	1	2014
9	F2FS	File recovers to incorrect size*	1	2015
10	F2FS	Persisted file disappears*	2	2016

Table 5: **Newly discovered bugs**. The table shows the new bugs found by CRASHMONKEY and ACE. The bugs have severe consequences, ranging from losing allocated blocks to entire files and directories disappearing. The bugs have been present for several years in the kernel, showing the need for systematic testing. Note that even workloads with single file-system operation have resulted in bugs. Developers have submitted a patch for bugs marked with *.

6.3 CrashMonkey Performance

CRASHMONKEY has three phases of operation: profiling the given workload, constructing crash states, and testing crash-consistency. Given a workload, the end-to-end latency to generate a bug report is 4.6 seconds. The main bottleneck is the kernel itself: mounting a file system requires up-to a second of delay (if CRASHMONKEY checks file-system state earlier, it sometimes gets an error). Similarly, once the workload is done, we also wait for two seconds to ensure the storage subsystem has processed the writes, and that we can unmount the file system without affecting the writes. These delays account for 84% of the time spent profiling.

After profiling, constructing crash states is relatively fast: CRASHMONKEY only requires 20 ms to construct each crash state. Furthermore, since CRASHMONKEY uses fine-grained correctness tests, checking crash consistency with both read and write tests takes only 20 ms.

6.4 Ace Performance

ACE generated all the workloads that were tested (3.37M) in 374 minutes ($\approx$ 150 workloads generated per second). Despite this high cost, it is important to note that generating workloads is a one-time cost. Once the workloads are generated, CRASHMONKEY can test these workloads on different file systems without any reconfiguration.

Deploying these workloads to the 780 virtual machines on Chameleon took 237 minutes: 34 minutes to group the workloads by virtual machines, 199 minutes to copy workloads to the Chameleon nodes, and 4 minutes to copy workloads to the virtual machines on each node.

These numbers reflect the time taken for a single local server to generate and push the workloads to Chameleon. By utilizing more servers and employing a more sophisticated strategy for generating workloads, we could reduce the time required to generate and push workloads.

6.5 Resource Consumption

The total memory consumption by CRASHMONKEY averaged across 10 randomly chosen workloads and the three sequence lengths is 20.12 MB. The low memory consumption results from the copy-on-write nature of the wrapper block device. Since ACE's workloads typically modify small amounts of data or metadata, the modified pages are few in number, resulting in low memory consumption. Furthermore, CRASHMONKEY uses persistent storage only for storing the workloads (480 KB per workload). Finally, the CPU consumption of CRASHMONKEY, as reported by `top`, was negligible (less than 1 percent).

7 Related Work

B^3 offers a new point in the spectrum of techniques addressing file-system crash consistency, alongside verified file systems and model checking. We now place B^3 in the context of prior approaches.

Verified File Systems. Recent work focuses on creating new, verified file systems from a specification [8, 9, 53]. These file systems are proven to have strong crash-consistency guarantees. However, the techniques employed are not useful for testing the crash consistency of existing, widely-used Linux file systems written in low-level languages like C. The B^3 approach targets such file systems, which are not amenable to verification.

Formal Crash-Consistency Models. Ferrite [6] formalizes crash-consistency models and can be used to test if a given ordering relationship holds in a file system; however, it is hard to determine what relationships to test. The authors used Ferrite to test a few simple relationships such as prefix append. On the other hand, ACE and CRASHMONKEY explore a wider range of workloads, and use oracles and developer-provided guarantees to automatically test correctness after a crash.

Model Checking. B^3 is closely related to in-situ model checking approaches such as EXPLODE [63] and FiSC [64]. However, unlike B^3, EXPLODE and FiSC require modifications to the buffer cache (to see all orderings of IO requests) and changes to the file-system code to expose choice points for efficient checking, a complex and time-consuming task. B^3 does not require changing any file-system code and it is conceptually simpler than in-situ model checking approaches, while still being effective at finding crash-consistency bugs.

Though the B^3 approach does not have the guarantees of verification or the power of model checking, it has the advantage of being easy to use (due to its black-box nature), being able to systematically test file systems (due to its exhaustive nature), and being able to catch crash-consistency bugs occurring on mature file systems.

Fuzzing. The B^3 approach bears some similarity to fuzz-testing techniques which explore inputs that will reveal bugs in the target system. The effectiveness of fuzzers is determined by the careful selection of uncommon inputs that would trigger exceptional behavior. However, B^3 does not randomize input selection. Neither does it use any sophisticated strategy to select workloads to test. Instead, B^3 exhaustively generates workloads in a bounded space, with the bounds informed by our study or provided by the user. While there exists fuzzers to test the correctness of system calls [17, 22, 45], there seem to be no fuzzing techniques to expose crash-consistency bugs. The effort by Nossum and Casasnovas [45] is closest to our work, where they generate file-system images that are likely to expose bugs during the normal operation of the file system (non-crash-consistency bugs).

Record and Replay Frameworks. CRASHMONKEY is similar to prior record-and-replay frameworks such as `dm-log-writes` [4], Block Order Breaker [47], and work by Zheng *et al.* [70]. Unlike `dm-log-writes`, which requires manual correctness tests or running `fsck`, CRASHMONKEY is able to automatically test crash-consistency in an efficient manner.

Similar to CRASHMONKEY, the Block Order Breaker (BOB) [47] also creates crash states from recorded IO. However, BOB is only used to show that different file systems persist file-system operations in significantly different ways. The Application-Level Intelligent Crash Explorer (ALICE), explores application-level crash vulnerabilities in databases, key value stores *etc.* The major drawback with ALICE and BOB is that they require the user to handcraft workloads and provide an appropriate checker for each workload. They lack systematic exploration of the workload space and do not understand persistence points, making it is extremely hard for a user to write bug-triggering workloads manually.

The logging and replay framework from Zheng *et al.* [70] is focused on testing whether databases provide ACID guarantees, works only on iSCSI disks, and uses only four workloads. CRASHMONKEY is able to test millions of workloads, and ACE allows us to generate a much wider ranger of workloads to test.

We previewed the ideas behind CRASHMONKEY in a workshop paper [36]. Since then, several features have been added to CRASHMONKEY with the prominent one being automatic crash-consistency testing.

8 Conclusion

This paper presents Bounded Black-Box Crash Testing (B^3), a new approach to testing file-system crash consistency. We study 26 crash-consistency bugs reported in Linux file systems over the past five years and find that most reported bugs could be exposed by testing small workloads in a systematic fashion. We exploit this insight to build two tools, CRASHMONKEY and ACE, that systematically test crash consistency. Running for two days on a research cluster of 65 machines, CRASHMONKEY and ACE reproduced 24 known bugs and found 10 new bugs in widely-used Linux file systems.

We have made CRASHMONKEY and ACE available (with demo, documentation, and a single line command to run `seq-1` workloads) at https://github.com/utsaslab/crashmonkey. We encourage developers and researchers to test their file systems against the workloads included in the repository.

Acknowledgments

We would like to thank our shepherd, Angela Demke Brown, the anonymous reviewers, and the members of Systems and Storage Lab and LASR group for their feedback and guidance. We would like to thank Sonika Garg, Subrat Mainali, and Fabio Domingues for their contributions to the CrashMonkey codebase. This work was supported by generous donations from VMware, Google, and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not reflect the views of other institutions.

References

[1] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers. Evolving ext4 for shingled disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 105–120, Santa Clara, CA, 2017. USENIX Association.

[2] Amazon. Amazon ec2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[3] Apple. fsync(2) mac os x developer tools manual page. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/fsync.2.html.

[4] J. Bacik. dm: log writes target. https://www.redhat.com/archives/dm-devel/2014-December/msg00047.html.

[5] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 69–86. ACM, 2017.

[6] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In T. Conte and Y. Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 83–98. ACM, 2016.

[7] btrfs Wiki. btrfs check. https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-check.

[8] H. Chen, T. Chajed, A. Konradi, S. Wang, A. Ileri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 270–286. ACM, 2017.

[9] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 18–37. ACM, 2015.

[10] V. Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[11] V. Chidambaram. btrfs: strange behavior (possible bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77929.html, Apr 2018.

[12] V. Chidambaram. btrfs: symlink not persisted even after fsync. https://www.spinics.net/lists/fstests/msg09379.html, Apr 2018.

[13] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[14] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, Feb. 2012.

[15] D. Chinner. btrfs: symlink not persisted even after fsync. https://www.spinics.net/lists/fstests/msg09363.html, Apr 2018.

[16] J. Corbet. Toward better testing. https://lwn.net/Articles/591985/, 2014.

[17] D. Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.

[18] T. O. Group. The open group base specifications issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[19] E. Guan. ext4: update idisksize if direct write past ondisk size. https://marc.info/?l=linux-ext4&m=151669669030547&w=2, Jan 2018.

[20] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[21] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *The 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, 2018. USENIX Association.

[22] D. Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium, pages*, 2011.

[23] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[24] U. B. LaunchPad. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.

[25] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[26] J. Mambretti, J. Chen, and F. Yeh. Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn). In *Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on*, pages 73–79. IEEE, 2015.

[27] F. Manana. btrfs: fix directory recovery from fsync log. https://patchwork.kernel.org/patch/4864571/, Sep 2014.

[28] F. Manana. btrfs: add missing inode update when punching hole. https://patchwork.kernel.org/patch/5830801/, Feb 2015.

[29] F. Manana. btrfs: fix fsync data loss after adding hard link to inode. https://patchwork.kernel.org/patch/5822681/, Feb 2015.

[30] F. Manana. btrfs: fix metadata inconsistencies after directory fsync. https://patchwork.kernel.org/patch/6058101/, March 2015.

[31] F. Manana. btrfs: fix stale directory entries after fsync log replay. https://patchwork.kernel.org/patch/6852751/, July 2015.

[32] F. Manana. btrfs: blocks allocated beyond eof are lost. https://www.spinics.net/lists/linux-btrfs/msg75108.html, Feb 2018.

[33] F. Manana. btrfs: fix log replay failure after unlink and link combination. https://www.spinics.net/lists/linux-btrfs/msg75204.html, Feb 2018.

[34] F. Manana. btrfs: strange behavior (possible bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg81425.html, Aug 2018.

[35] F. Manana. btrfs: sync log after logging new name. https://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg77875.html, Jun 2018.

[36] A. Martinez and V. Chidambaram. Crashmonkey: a framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, pages 6–6. USENIX Association, 2017.

[37] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[38] M. K. McKusick, G. R. Ganger, et al. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, 1999.

[39] R. McMillan. Amazon Blames Generators For Blackout That Crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazonexplains/, 2012.

[40] R. Miller. Power Outage Hits London Data Center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/, 2012.

[41] R. Miller. Data Center Outage Cited In Visa Downtime Across Canada. http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/, 2013.

[42] R. Miller. Power Outage Knocks Dreamhost Customers Offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/, 2013.

[43] J. Mohan. btrfs: hard link not persisted on fsync. https://www.spinics.net/lists/linux-btrfs/msg76878.html, Apr 2018.

[44] J. Mohan. btrfs: inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77219.html, Apr 2018.

[45] V. Nossum and Q. Casasnovas. Filesystem fuzzing with american fuzzy lop. https://lwn.net/Articles/685182/, 2016.

[46] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 181–196, Santa Clara, CA, 2017. USENIX Association.

[47] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[48] POSIX. fsync: The open group base specifications issue 6. http://pubs.opengroup.org/onlinepubs/009695399/functions/fsync.html.

[49] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[50] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J. Hwang, S. Cho, D. D. Lee, and J. Jeong. Fstream: Managing flash streams in the file system. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 257–264, Oakland, CA, 2018. USENIX Association.

[51] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[52] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[53] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 1–16, Berkeley, CA, USA, 2016. USENIX Association.

[54] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, 2018. USENIX Association.

[55] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, Jan. 1996.

[56] T. Y. Ts'o. btrfs: Inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77389.html, Apr 2018.

[57] T. Y. Ts'o. btrfs: Inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77340.html, Apr 2018.

[58] UTSASLab. Crash-consistency bugs studied and reproduced. https://github.com/utsaslab/crashmonkey/blob/master/reproducedBugs.md.

[59] UTSASLab. New crash-consistency bugs found. https://github.com/utsaslab/crashmonkey/blob/master/newBugs.md.

[60] J. Verge. Internap Data Center Outage Takes Down Livestream And Stackexchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[61] R. S. V. Wolffradt. Fire In Your Data Center: No Power, No Access, Now What? http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html, 2014.

[62] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 211–226, Berkeley, CA, USA, 2018. USENIX Association.

[63] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, Nov. 2006.

[64] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors (awarded best paper!). In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 273–288. USENIX Association, 2004.

[65] C. Yu. f2fs: keep isize once block is reserved cross eof. https://sourceforge.net/p/linux-f2fs/mailman/message/36104201/, Nov 2017.

[66] C. Yu. f2fs: enforce fsync_mode=strict for renamed directory. https://lkml.org/lkml/2018/4/25/674, Apr 2018.

[67] C. Yu. f2fs: fix to set keep_size bit in f2fs_zero_range. https://lore.kernel.org/patchwork/patch/889955/, Feb 2018.

[68] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 1–14, Santa Clara, CA, 2016. USENIX Association.

[69] S. Zhang, H. Catanese, and A. A.-I. Wang. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 15–22, Santa Clara, CA, 2016. USENIX Association.

[70] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, 2014. USENIX Association.

An Analysis of Network-Partitioning Failures in Cloud Systems

Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present a comprehensive study of 136 system failures attributed to network-partitioning faults from 25 widely used distributed systems. We found that the majority of the failures led to catastrophic effects, such as data loss, reappearance of deleted data, broken locks, and system crashes. The majority of the failures can easily manifest once a network partition occurs: They require little to no client input, can be triggered by isolating a single node, and are deterministic. However, the number of test cases that one must consider is extremely large. Fortunately, we identify ordering, timing, and network fault characteristics that significantly simplify testing. Furthermore, we found that a significant number of the failures are due to design flaws in core system mechanisms.

We found that the majority of the failures could have been avoided by design reviews, and could have been discovered by testing with network-partitioning fault injection. We built NEAT, a testing framework that simplifies the coordination of multiple clients and can inject different types of network-partitioning faults. We used NEAT to test seven popular systems and found and reported 32 failures.

1 Introduction

With the increased dependency on cloud systems [1, 2, 3, 4], users expect high—ideally, 24/7—service availability and data durability [5, 6]. Hence, cloud systems are designed to be highly available [7, 8, 9] and to preserve data stored in them despite failures of devices, machines, networks, or even entire data centers [10, 11, 12].

Our goal is to better understand the impact of a specific type of infrastructure fault on modern distributed systems: network-partitioning faults. We aim to understand the specific sequence of events that lead to user-visible system failures and to characterize these system failures to identify opportunities for improving system fault tolerance.

We focus on network partitioning for two reasons. The first is due to the complexity of tolerating these faults [13, 14, 15, 16]. Network-partitioning fault tolerance pervades the design of all system layers, from the communication middleware and data replication [13, 14, 16, 17] to user API definition and semantics [18, 19], and it dictates the availability and consistency levels a system can achieve [20]. Second, recent studies [21, 22, 23, 24] indicate that, in production networks, network-partitioning faults occur as frequently as once a week and take from tens of minutes to hours to repair.

Given that network-partitioning fault tolerance is a well-studied problem [13, 14, 17, 20], this raises questions about *how these faults sill lead to system failures. What is the impact of these failures? What are the characteristics of the sequence of events that lead to a system failure? What are the characteristics of the network-partitioning faults? And, foremost, how can we improve system resilience to these faults?*

To help answer these questions, we conducted a thorough study of 136 network-partitioning failures[1] from 25 widely used distributed systems. The systems we selected are popular and diverse, including key-value systems and databases (MongoDB, VoltDB, Redis, Riak, RethinkDB, HBase, Aerospike, Cassandra, Geode, Infinispan, and Ignite), file systems (HDFS and MooseFS), an object store (Ceph), a coordination service (ZooKeeper), messaging systems (Kafka, ActiveMQ, and RabbitMQ), a data-processing framework (Hadoop MapReduce), a search engine (Elasticsearch), resource managers (Mesos, Chronos, and DKron), and in-memory data structures (Hazelcast, Ignite, and Terracotta).

For each considered failure, we carefully studied the failure report, logs, discussions between users and developers, source code, code patch, and unit tests. We manually reproduced 24 of the failures to understand the specific manifestation sequence of the failure.

Failure impact. Overall, we found that network-partitioning faults lead to *silent catastrophic* failures (e.g., data loss, data corruption, data unavailability, and broken locks), with *21% of the failures leaving the system in a lasting erroneous state* that persists even after the partition heals.

Ease of manifestation. Oddly, it is easy for these failures to occur. *A majority of the failures required three or fewer frequently used events (e.g., read, and write), 88% of them can be triggered by isolating a single node, and 62% of them were deterministic.* It is surprising that catastrophic failures manifest easily, given that these systems are generally developed using good software-engineering practices and are subjected to multiple design and code reviews as well as thorough testing [5, 25].

[1] A *fault* is the initial root cause, including machine and network problems and software bugs. If not properly handled a fault may lead to a user-visible system *failure*.

Partial Network Partitions. Another unexpected result is that a *significant number of the failures (29%) were caused by an unanticipated type of fault: partial network partitions.* Partial partitions isolate a set of nodes from some, but not all, nodes in the cluster, leading to a confusing system state in which the nodes disagree whether a server is up or down. The effects of this disagreement are poorly understood and tested. This is the first study to analyze the impact of this fault on modern systems.

Testability. We studied the testability of these failures. In particular, we analyzed the manifestation sequence of each failure, ordering constraints, timing constraints, and network fault characteristics. While the number of event permutations that can lead to a failure is excessively large, we identified characteristics that significantly reduce the number of test cases (Section 5). We also found that *the majority of the failures can be reproduced through tests and by using only three nodes.*

Our findings debunk two common presumptions. First, network practitioners presume that systems, with their software and data redundancy, are robust enough to tolerate network partitioning [22]. Consequently, practitioners assign low priority to the repair of top-of-the-rack (ToR) switches [22], even though these failures isolate a rack of machines. Our findings show that this presumption is ill founded, as *88% of the failures can occur by isolating a single node.* Second, system designers assume that limiting client access to one side of a network partition will eliminate the possibility of a failure [28, 29, 30, 31, 32, 33, 34]. Our findings indicate that *64% of the failures required no client access at all or client access to only one side of the network partition.*

We examined the unit tests that we could relate to the studied code patches and we found that developers lack the proper tools to test these failures. In most cases, developers used mocking [26, 27] to test the impact of network partitioning on only one component and on just one side of the partition. However, this approach is inadequate for end-to-end testing of complete distributed protocols.

Our findings motivated us to build the network partitioning testing framework (NEAT). NEAT simplifies testing by allowing developers to specify a global order for client operations and by providing a simple API for creating and healing partitions as well as crashing nodes. NEAT uses OpenFlow [35] to manipulate switch-forwarding rules and create partitions. For deployments that do not have an OpenFlow switch, we built a basic version using `iptables` [36] to alter firewall rules at end hosts.

We used NEAT to test seven systems: Ceph [37], ActiveMQ [38], Apache Ignite [39], Terracotta [40], DKron [41], Infinispan [42], and MooseFS [43]. We found and reported 32 failures that led to data loss, stale reads, reappearance of deleted data, unavailability, double locking, and broken locks.

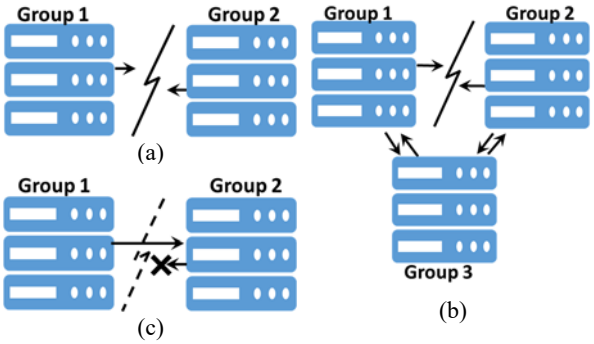


Figure 1. Network partitioning types. (a) Complete partition: The system is split into two disconnected groups (b) Partial partition: The partition affects some, but not all, nodes in the system. Group 3 in Figure (b) can communicate with the other two groups. (c) Simplex partition, in which traffic flows only in one direction.

The rest of this paper is organized as follows: In Section 2, we present a categorization of network-partitioning faults, discuss the theoretical limit on system design, and discuss the current testing techniques. In Section 3, we present our methodology and its limitations. Then, we present our findings in Sections 4 and 5 and discuss a number of related observations in Section 6. We present the NEAT framework in Section 7. We present additional related work in Section 8. We share our insights in Section 9 and conclude our paper in Section 10.

2 Background

In this section, we present the three types of network-partitioning faults (Section 2.1), discuss the theoretical limit for systems design (Section 2.2), and survey the current approaches for testing systems' resilience to network-partitioning faults (Section 2.3).

2.1 Types of Network Partitions

Modern networks are complex. They span multiple data centers [44, 45], use heterogeneous hardware and software [23], and employ a wide range of middle boxes (e.g., NAT, load balancers, route aggregators, and firewalls) [21, 44, 45]. Despite the high redundancy built into modern networks, catastrophic failures are common [21, 22, 23, 24]. We surveyed network-partitioning failures and identified three types:

Complete network partitioning leads to dividing the network into two disconnected parts (Figure 1.a). Complete partitions can happen at different scales; for example, they can manifest in geo-replicated systems due to the loss of connectivity between data centers. HP reported that 11% of its enterprise network failures lead to site connectivity problems [23]. Turner et al. found that a network partition occurs almost once every 4

days in the California-wide CENIC network [24]. In a data center, a complete partition can manifest due to failures in the core or aggregation switches [22] or because of a ToR switch failure. Microsoft and Google report that ToR failures are common and have led to 40 network partitions in two years at Google [21] and caused 70% of the downtime at Microsoft [22]. Finally, NIC failures [46] or bugs in the networking stack can lead to the isolation of a single node that could be hosting multiple VMs. Finally, network-partition faults caused by correlated failures of multiple devices are not uncommon [22, 24, 44]. Correlated switch failures are frequently caused by system-wide upgrades and maintenance tasks [21, 22].

Partial network partitioning is a fault that leads to the division of nodes into three groups (Group1, Group2, and Group3 in Figure 1.b) such that two groups (say, Group1 and Group2) are disconnected while Group3 can communicate with both Group1 and Group2 (Figure 1.b). Partial partitions are caused by a loss of connectivity between two data centers [23] while both are reachable by a third center, or due to inconsistencies in switch-forwarding rules [21].

Simplex network partitioning permits traffic to flow in one direction, but not in the other (Figure 1.c). This is the least common failure and can be caused by inconsistent forwarding rules or hardware failures (e.g., the Broadcom BCM5709 chipset bug [46]). The impact of this failure is mainly manifested in UDP-based protocols. For instance, a simplex network partitioning dropped all incoming packets to a primary server while allowing the primary server heartbeats to reach the failover server. The system hang as the failover server neither detected the failure nor took over [46].

2.2 Theoretical Limit

The data consistency model defines which values a read operation may return. The strong consistency model [47] (a.k.a. sequential consistency) is the easiest to understand and use. Strong consistency promises that a read operation will return the most recent successfully written value. Unfortunately, providing strong consistency reduces system availability and requires complex consistency protocols [13, 14, 17]. Gilbert and Lynch [20] presented a theoretical limit on system design. Their theorem, famously known as the CAP theorem, states that in the presence of a network partition, designers need to choose between keeping the service available and maintaining data consistency.

To maintain system availability, system designers choose a relaxed consistency model such as the read-your-write [11, 18, 19, 48], timeline [19, 48, 49], and eventually consistent [16, 19, 50, 51] models.

Modern systems often implement consensus protocols that have not been theoretically proven. Eventually consistent systems implement unproven protocols (Hazelcast [29] and Redis [32]), and systems that implement proven, strongly consistent protocols (e.g., Paxos [13] and Raft [14]) often tweak these protocols in unproven ways [15, 31, 52]. These practices make modern systems vulnerable to unforeseen failure scenarios, such as the ones caused by different types of network partitions.

2.3 Testing with Network Partitioning

A common testing technique for network-partitioning failures is mocking. Mocking frameworks (e.g., Mockito [26]) can be used to imitate communication problems. Mocking can be employed to test the impact of a failure on a single component, but it is not suitable for system-level testing or for testing distributed protocols. A few systems use hacks to emulate a network partition; for instance, Mesos' unit tests emulate a network partition by ignoring test-specific messages received by the protobuf middleware [53].

Another possible testing approach is to use the Jepsen testing framework [54]. Jepsen is written in Clojure [55] and is tuned toward random testing. Jepsen testing typically involves running an auto-generated testing workload while the tool injects network-partitioning faults. Jepsen does not readily support unit testing or all types of network partitioning.

We built NEAT, a Java-based, system-level testing framework. NEAT has a simple API for deploying systems, specifying clients' workloads, creating and healing partitions, and crashing nodes. Unlike Jepsen, NEAT readily supports injecting the three types of network-partitioning faults.

3 Methodology and Limitations

We studied 136 real-world failures in 25 popular distributed systems. We selected a diverse set of distributed systems (Table 1), including 10 key-value storage systems and databases, a coordination service, two file systems, an object storage system, three message-queueing systems, a data-processing framework, a search engine, three resource managers, and three distributed in-memory caches and data structures. We selected this diverse set of systems to understand the wide impact of network-partitioning faults on distributed systems and because these systems are widely used and are considered production quality.

The 136 failures[2] we studied include 88 failures extracted from the publicly accessible issue-tracking systems, 16 Jepsen reports [54], and 32 failures detected by our NEAT framework (Section 7). The majority of the studied tickets contain enough details to

[2] We differentiate failures by their manifestation sequence of events. In a few cases, the same faulty mechanism leads to two different failures and impacts depending on workload. We count these as separate failures, even if they were reported in a single ticket. Similarly, although the exact failure is sometimes reported in multiple tickets, we count it once in our study.

Table 1. List of studied system. The table shows systems' consistency model, number of failures, and number of catastrophic failures. Highlighted rows indicate systems we tested using NEAT, and the number of failures we found.

System	Consistency Model	Failures	
		Total	Catastrophic
MongoDB [31]	Strong	19	11
VoltDB [33]	Strong	4	4
RethinkDB [52]	Strong	3	3
HBase [56]	Strong	5	3
Riak [57]	Strong/Eventual	1	1
Cassandra [58]	Strong	4	4
Aerospike [59]	Eventual	3	3
Geode [60]	Strong	2	2
Redis [32]	Eventual	3	2
Hazelcast [29]	Best Effort	7	5
Elasticsearch [28]	Eventual	22	21
ZooKeeper [61]	Strong	3	3
HDFS [1]	Custom	4	2
Kafka [30]	-	5	3
RabbitMQ [62]	-	7	4
MapReduce [4]	-	6	2
Chronos [63]	-	2	1
Mesos [64]	-	4	0
Infinispan [42]	Strong	1	1
Ignite [39]	Strong	15	13
Terracotta [40]	Strong	9	9
Ceph [37]	Strong	2	2
MooseFS [43]	Eventual	2	2
ActiveMQ [38]	-	2	2
DKron [41]	-	1	1
Total	-	**136**	**104**

understand the failure. These tickets document failures that were confirmed by the developers and include discussions between the users and the developers, steps to reproduce the failure, outputs and logs, code patch, and sometimes unit tests.

The 88 failures we included in our study were selected as follows: First, we used the search tool in the issue-tracking systems to identify tickets related to network partitioning. We searched using the following keywords: "network partition," "network failure," "switch failure," "isolation," "split-brain," and "correlated failures." Second, we considered tickets that were dated 2011 or later. Third, we excluded low-priority tickets that were marked as "Minor" or "Trivial." Fourth, we examined the set of tickets to verify that they were indeed related to network-partitioning failures and excluded tickets that appeared to be part of the development cycle; for instance, they discuss a feature design. Finally, some failures that are triggered by a node crash can also be triggered by a network partition isolating that node. We excluded failures that can be triggered by a node crash and studied failures that can only be triggered by a network partition. Out of all Jepsen blog posts (there is 25 in total), we included 16 that are related to the systems we studied. Table 1 shows the number of failures and the consistency model of the systems we studied.

For each ticket, we studied the failure description, system logs, developers' and users' comments, code patch, and unit tests. Using NEAT, we also reproduced 13 failures reported in the issue-tracking systems, as well as 11 failures reported by Jepsen to understand their intricate details.

Limitations: As with any characterization study, there is a risk that our findings may not be generalizable. Here we list three potential sources of bias and describe our best efforts to address them.

1) *Representativeness of the selected systems*. Because we only studied 25 systems, the results may not be generalizable to the hundreds of systems we did not study. However, we selected a diverse set of systems (Table 1). These systems follow diverse designs, from persistent storage and reliable in-memory storage to volatile caching systems. They use leader-follower or peer-to-peer architectures; are written in Java, C, Scala, or Erlang; adopt strong or eventual consistency; use synchronous or asynchronous replication; and use chain or parallel replication. The systems we selected are widely used: ZooKeeper is a popular coordination service; Kafka is the most popular message-queueing system; MapReduce, HDFS, and HBase are the core of the dominant Hadoop data analytics platform; MongoDB, Riak, Aerospike, Redis, and VoltDB are popular key-value-based databases; and Hazelcast, Ignite, and Terracotta are popular tools in a growing area of in-memory distributed data structures.
2) *Sampling bias*. The way we choose the tickets may bias the results. We designed our methodology to include high impact tickets. Modern systems take node unreachability as an indicator of a node crash. Consequently, a network partition that isolates a single node can trigger the same failures that are caused by a single node crash. We excluded failures that can be caused by a node crash and considered those that are solely triggered by a network partitioning fault (i.e., the nodes on both sides of the partition must be running for a failure to manifest). Furthermore, we eliminated all low-priority tickets and focused on tickets the developers considered important. All presented findings should be interpreted with this sampling methodology in mind.
3) *Observer error*. To minimize the possibility of observer errors, all failures were independently reviewed by two team members and discussed in a group meeting before agreement was reached, and all team members used the same detailed classification methodology.

4 General Findings

This section presents the general findings from our study. Overall, our study indicates that network partitioning leads to catastrophic failures. However, it identifies failure characteristics that can improve testing. We show that most of the studied failures can be reproduced using only three nodes and are deterministic or have bounded timing constraints. We show that core distributed system mechanisms are the most vulnerable, including leader election, replication, and request routing. Finally, we show that a large number of the failures are caused by partial network-partitioning faults.

4.1 Failure Impact

Overall, our findings indicate that network-partitioning faults cause silent catastrophic failures that can result in lasting damage to systems.

Finding 1. *A large percentage (80%) of the studied failures have a catastrophic impact, with data loss being the most common (27%)* (Table 2).

We classify a failure as catastrophic if it violates the system guarantees or leads to a system crash. Table 2 lists the different types of catastrophic failures. Failures that degrade performance or crash a single node are not considered catastrophic. Stale reads are catastrophic only when the system promises strong consistency. However, they are not considered failures in eventually consistent systems. Dirty reads happen when the system returns the value of a preceding unsuccessful write operation. For instance, a client reading from the primary replica in MongoDB may get a value that is simultaneously being written by a concurrent write operation [65]. If the write fails due to network partitioning, the read operation has returned a value that was never successfully written (a.k.a. dirty read).

Compared to other causes of failures, this finding indicates that network partitioning leads to a significantly higher percentage of catastrophic failures. Yuan et al. [66] present a study of 198 randomly selected, high-priority failures from five of the systems we include in our study: Cassandra, HBase, HDFS, MapReduce, and Redis. They report that only 24% of failures had catastrophic effects[3], compared to 80% in the case of network-partitioning failures (Table 2). Consequently, developers should carefully consider this fault in all phases of system design, development, and testing.

Table 2. The impacts of the failures. The percentage of the failures that cause each impact. Broken locks include double locking, lock corruption, and failure to unlock.

Impact	%	
Data loss	26.6%	Catastrophic (79.5%)
Stale read	13.2%	
Broken locks	8.2%	
System crash/hang	8.1%	
Data unavailability	6.6%	
Reappearance of deleted data	6.6%	
Data corruption	5.1%	
Dirty read	5.1%	
Performance degradation	19.1%	
Other	1.4%	

Finding 2. *The majority (90%) of the failures are silent, whereas the rest produce warnings that are unatonable.*

We inspected the failure reports for returned error messages and warnings. The majority of the failures were silent (i.e., no error or warning was returned to the client), with some failures (10%) returning warning messages to the client. Unfortunately, all returned warnings were confusing, with no clear mechanism for resolution. For instance, in Riak [67] with a strict quorum configuration, when a write fails to fully replicate a new value, the client gets a warning indicating that the write operation has updated a subset of replicas, but not all of them. This warning is confusing because it does not indicate the necessary action to take next. Similarly, MongoDB returns a generic socket exception if a proxy node cannot reach the data nodes [68].

This is alarming because users and administrators are not notified when a failure occurs, which delays failure discovery, if the failure is discovered at all.

Finding 3. *Twenty one percent of the failures lead to permanent damage to the system. This damage persists even after the network partition heals.*

While 79% of the failures affect the system only while there is a network partition, 21% of the failures leave the system in an erroneous state that persists even after the network partition heals. For instance, if a new node is unable to reach the other nodes in RabbitMQ [69] and Ignite (section 7.4), the node will assume that the rest of the cluster has failed and will form a new independent cluster. These clusters will remain separated, even after the network partition heals.

Overall, as recent studies [21, 22, 23, 24] indicate that network-partitioning faults occur as frequently as once a week and take from tens of minutes to hours to repair, it is alarming that these faults can lead to silent catastrophic failures. This is surprising, given that these systems are designed for deployments in which component failure is the norm. For instance, all of the systems we studied replicate their data. In MongoDB, Hazelcast, Kafka, Elasticsearch, Geode, Mesos, Redis,

[3] We note that these percentages are not directly comparable as our definition of catastrophic failure is more conservative. For instance, while Yuan et al. [66] count a loss of a single replica or a crash of a single node as catastrophic, we do not.

VoltDB, and RethinkDB, if a leader node is partitioned apart from the majority, then the rest of the nodes will quickly elect a new leader. Hazelcast and VoltDB employ "split-brain protection," a technique that continuously monitors the network and pauses nodes in the minority partition if a network partition is detected. Furthermore, ZooKeeper and MongoDB include a mechanism for data consolidation. How, then, do these failures still occur?

4.2 Vulnerability of System Mechanisms

Finding 4. *Leader election, configuration change, request routing, and data consolidation are the most vulnerable mechanisms to network partitioning* (Table 3).

Leader election is the most vulnerable to network partitioning (was affected by 40% of the failures). We further analyzed leader election failures (Table 4) and found that the most common leader election flaw is the simultaneous presence of two leaders. This failure typically manifests as follows: A network partition isolates the current leader from the majority of replicas. The majority partition elects a new leader. The old leader may eventually detect that it no longer has a majority of replicas at its side and step down. However, there is a period of time in which each network partition has a leader. The overlap between the two leaders may last until the network partition heals (which may take hours [21]). In MongoDB [70], VoltDB [71], and Raft-based RethinkDB [72], if a network partition isolates a leader, the isolated leader will not be able to update the data, but it will still respond to read requests from its local copy, leading to stale and dirty reads.

In all of the systems we studied, the leader trusts that its data set or log is complete and all replicas should update/trim their data sets to match the leader copy. Consequently, it is critical to elect the leader with a complete and consistent data set. Table 4 shows that 20% of leader election failures are caused by electing a bad leader. This is caused by using simple criteria for leader election, such as the node with the longest log wins (e.g., VoltDB), the node that has the latest operation timestamp wins (e.g., MongoDB), or the node with the lowest id wins (e.g., Elasticsearch). These criteria can cause data loss when a node from the minority partition becomes a leader and erases all updates performed by the majority partition.

Conflicting election criteria lead to 3.7% of the leader election failures and are only reported in MongoDB. MongoDB leader election has multiple criteria for electing a leader. One can assign a priority for a replica to become a leader. The priority node will reject any leader proposal; similarly, the node with the latest operation timestamp will reject all leader proposals, leaving the cluster without a leader [73].

Table 3. The percentage of the failures involving each system mechanism. Some failures involve multiple mechanisms.

Mechanism		%
Leader election		39.7%
Configuration change		19.9%
▪ Adding a node	10.3%	
▪ Removing a node	3.7%	
▪ Membership management	3.7%	
▪ Other	2.2%	
Data consolidation		14.0%
Request routing		13.2%
Replication protocol		12.5%
Reconfiguration due to a network partition		11.8%
Scheduling		2.9%
Data migration		3.7%
System integration		1.5%

Table 4. Leader election flaws.

Leader election failure	%
Overlapping between successive leaders	57.4%
Electing bad leaders	20.4%
Voting for two candidates	18.5%
Conflicting election criteria	3.7%

The second most affected mechanism is configuration change, including node join or leave and role changes (e.g., changing the primary replica). We discuss two examples of these failures in Section 4.4.

The third most affected mechanism is data consolidation. Failures in this mechanism typically lead to data loss in both eventually and strongly consistent systems. For instance, Redis, MongoDB, Aerospike, Elasticsearch, and Hazelcast employ simple policies to automate data consolidation, such as the write with the latest timestamp wins and the log with the most entries wins. However, because these policies do not check the replication or operation status, they can lose data that is replicated on the majority of nodes and that was acknowledged to the client.

The three ZooKeeper failures that we studied are related to data consolidation. For instance, ZooKeeper has two mechanisms for synchronizing data between nodes: storage synchronization that is used for syncing a large amount of data, and in-memory log synchronization that is used for a small amount of data. If node A misses many updates during a network partition, then ZooKeeper will use storage synchronization to bring node A up to date. Unfortunately, storage synchronization does not update the in-memory log. If A becomes a leader, and other nodes use in-memory log synchronization, then A will replicate its incomplete in-memory log [74].

Request routing represents the mechanism for routing requests or responses between clients and the specific nodes that can serve the request. Failures in request routing represent 13.2% of the failures. The

majority of those failures are caused by failing to return a response. For instance, in Elasticsearch, if a replica (other than the primary) receives write requests, it acts as a coordinator and forwards the requests to the primary replica. If a primary completes the write operation but fails to send an acknowledgment back to the coordinator, then the coordinator will assume the operation has failed and will return an error code to the client. The next client read will return the value written by a write operation that was reported to have failed. Moreover, if the client repeats the operation, then it will be executed twice [75].

The rest of the failures were caused by flaws in the replication protocol, scheduling, data migration mechanism, system integration with ZooKeeper, and system reconfiguration in response to network partitioning failures, in which the nodes remove the unreachable nodes from their replica set.

These findings are surprising because 15 of the systems use majority voting for leader election to tolerate exactly this kind of failure. Similarly, the primary purpose of a data consolidation mechanism is to correctly resolve conflicting versions of data. To improve resilience, this finding indicates that developers should enforce tests and design reviews focusing on network-partitioning fault tolerance, especially on these mechanisms.

4.3 Network Faults Analysis

Finding 5. *The majority (64%) of the failures either do not require any client access or require client access to only one side of the network partition* (Table 5).

This finding debunks a common presumption that network partitioning mainly leads to data conflicts, due to concurrent writes at both sides of the partition. Consequently, developers ensure that clients can only access one side of the partition to eliminate the possibility of a failure [28, 29, 30, 31, 32, 33, 34]. As an example of a failure that requires client access to one side of the partition, in HBase, region servers process client requests and store them in a log in HDFS. When the log reaches a certain size, a new log is created. If a partial partition separates a region server from the HMaster but not from HDFS, then the HMaster will assume that the region server has crashed and will assign the region logs to other servers. At this time, if the old region server creates a new log, HMaster will not be aware of the new log and will not assign it to any region server. All client operations stored in the new log will be lost [76]. We discuss a MapReduce failure that does not require any client access in section 4.4.

This finding indicates that system designers must consider the impact of a network partition fault on all system operations, including asynchronous client operations and offline internal operations.

Table 5. Percentage of the failures that require client access *during* the network partition

Client Access	%
No client access necessary	28%
Client access to one side only	36%
Client access to both sides	36%

Table 6. Percentage of the failures caused by each type of network-partitioning fault.

Partition type	%
Complete partition	69.1%
Partial partition	28.7%
Simplex partition	2.2%

Finding 6. *While the majority (69%) of the failures require a complete partition, a significant percentage of them (29%) are caused by partial partitions* (Table 6).

Partial network partitioning failures are poorly understood and tested, even by expert developers. For instance, most of the network-partitioning failures in Hadoop MapReduce and HDFS are caused by partial network-partitioning faults. In the following section, we discuss these failures in detail.

Simplex network partitioning caused 2% of the failures. This type of fault only confuses UDP-based protocols and leads to performance degradation. For instance, in HDFS [77], a data node that can send a periodic heartbeat message but is unable to receive requests is still considered a healthy node.

The overwhelming majority (99%) of the failures were caused by a single network partition. Only 1% of the failures required two network partitions to manifest.

4.4 Partial Network-Partitioning Failures

To the best of our knowledge, this the first study to analyze and highlight the impact of partial network partitions on systems. Consequently, we dedicate this section to discussing our insights and presenting detailed examples of how these failures manifest.

We found that the majority of partial network-partitioning failures are due to design flaws. This indicates that developers do not anticipate networks to fail in this way. Other than that, partial partitions failures had impact, ordering, and timing characteristics that are similar to complete partition failures.

Tolerating partial network partitions is complicated because these faults lead to inconsistent views of a system state; for instance, nodes disagree on whether a server is up or down. This confusion leads part of the system to carry on normal operations, while another part executes fault tolerance routines. Apparently, the mix of these two modes is poorly tested. The following are four examples:

- *Scheduling in MapReduce and Elasticsearch.* In MapReduce, if a partial network partition isolates an AppMaster from the resource manager while both

can still communicate with the cluster nodes, the AppMaster will finish executing the current task and return the result to the client. The resource manager will assume that the AppMaster has failed and will rerun the task using a new AppMaster. The new AppMaster will execute the task again and send a second result to the client. This failure will confuse the client and will lead to data corruption and double execution [78]. Note that in this failure, there is no client access after the network partition.

Elasticsearch has a similar failure [75]—if a coordinator does not get the result from a primary node, the coordinator will run the task again, leading to double execution.

- *Data placement in HDFS.* If a partial network partition separates a client from, say, rack 0, while the NameNode can reach that rack. If the NameNode allocates replicas on rack 0, then a client write operation will fail, and the client will ask for a different replica. The NameNode, following its rack-aware data placement, will likely suggest another node from the same rack. The process repeats five times before the client gives up [79].
- *Leader election in MongoDB and Elasticsearch.* MongoDB design includes an arbiter process that participates in a leader election to break ties. Assume a MongoDB cluster with two replicas (say A and B) and an arbiter, with A being the current leader. Assume a partial network partition separates A and B, while the arbiter can reach both nodes. B will detect that A is unreachable and will start a leader election process; being the only contestant, it will win the leadership. The arbiter will inform A to step down. After missing three heartbeats from the current leader (i.e., B), A will assume that B has crashed, start the leader election process, and become a leader. The arbiter will inform B to step down. This thrashing will continue until the network partition heals [80]. MongoDB does not serve client requests during leader election; consequently, this failure significantly reduces availability.

  Elasticsearch has a similar failure [81], in which a partial partition leads to having two simultaneous leaders because nodes that can reach the two partitions become followers of the two leaders. Note that these failures do not require any client access.
- *Configuration change in RethinkDB and Hazelcast.* RethinkDB is a strongly consistent database based on Raft [52]. Unlike Raft, when an admin removes a replica from RethinkDB cluster, the removed replica will delete its Raft log. This apparently minor tweak of the Raft protocol leads to a catastrophic failure. For instance, if a partial network partition breaks a replica set of five servers (A, B, C, D, and E) such that the (A, B) partition cannot reach (D, E) while C can reach all nodes, then if D receives a request to change the replication to two, D will remove A, B, and C from the set of replicas. C will delete its log. A and B will be unaware of the configuration change and still think that C is an active replica. C, having lost its Raft log that contains the configuration change request, will respond to A and B requests. This scenario creates two replica sets for the same keys. D and E are a majority in the new configuration, and A, B, and C are a majority in the old configuration [72].

  Hazelcast has a similar failure [82]. In Hazelcast, nodes delete their local data on configuration change. If a partial partition separates the new primary replica, then one replica will promote itself to become the primary. If the central master can reach both partitions, it will see that the old primary is still alive and inform the self-promoted replica to step down. That replica will step down, delete its data, and try to download the data from the primary. If the primary permanently fails before the partition heals, the data will be lost [82].

5 Failure Complexity

To understand the complexity of these failures, we studied their manifestation sequence, importance of input events order, network fault characteristics, timing constraints, and system scale. The majority of the failures are deterministic, require three or fewer input events, and can be reproduced using only three nodes. These characteristics indicate that it is feasible to test for these failures using limited resource.

5.1 Manifestation Sequence Analysis

Finding 7. *A majority (83%) of the failures triggered by a network partition require an additional three or fewer input events to manifest* (Table 7).

Table 8 lists the events that led to failures. All of the listed operations are frequently used. Read and write operations are part of over 50% of the failures, and 12.6% of the failures do not require any events other than a single network-partitioning fault. As an example of a failure without any client access, in Redis [83], if a network partition separates two nodes during a sync operation, the data log on the receiving node will be permanently corrupted. Similarly, in RabbitMQ [84], if a partial partition isolates one node from the leader, but not from the rest of the replicas, that node will assume the leader has crashed. The isolated node will become the new leader. When the old leader receives a notification to become a follower, it will start a follower thread but will not stop the leader thread. The contention between the follower and leader threads results in a complete system hang.

This is perilous, as a small number of frequently used events can lead to catastrophic failures.

Table 7. The minimum number of events required to cause a failure. The table counts a network-partitioning fault as an event. Note that 12.5% of the failures require no client access, neither during a network partition nor after it heals. Note that 28% of the failures reported in Table 5 do not require client access *during* the partition, but around 15.5% require client access before or after the network partition occurs.

Number of events	%
1 (just a network partition)	12.6%
2	13.9%
3	42.6%
4	14.0%
> 4	16.9%

Table 8. Percentage of faults each event is involved in.

Event type	%
Only a network-partitioning fault	12.6%
Write request	48.5%
Read request	34.6%
Acquire lock	8.1%
Admin adding/removing a node	8.0%
Delete request	4.4%
Release lock	3.7%
Whole cluster reboot	1.5%

Finding 8. *All of the failures that involve multiple events only manifest if the events happen in a specific order.*

All of the 87% of failures that require multiple events (2 events or more in Table 7) need the events to occur in a specific order. This implies that to expose these failures we not only need to explore the combination of these events, but also the different permutations of events, which makes the event space extremely large.

Fortunately, we identified characteristics that significantly prune this large event space and make testing tractable (Table 9). First, 84% of the manifestation sequences start with a network-partitioning fault. For 27.7% of the sequences, the order of the rest of events is not important, and in 27% of the sequences the events follow a natural order; that is, lock() comes before unlock(), and write() before read().

While this finding indicates that reproducing a failure can be complex, the probability of a failure in production is still high. The majority of multi-event failures require three or fewer events (Table 7); consequently, it is highly likely for a system that experiences a network partitioning for hours to receive all possible permutations of these common events.

Table 9. Ordering charactrisitcs.

Ordering Charactrisitcs		%
Network partition does not come first		16.0%
Network partition comes first		84.0%
▪ Order is not important	27.7%	
▪ Natural order	26.9%	
▪ Other	29.4%	

Table 10. System connectivity during the network partition. Examples of a central service include a ZooKeeper cluster and HBase master. Examples of nodes with a special role include MongoDB arbiter and MapReduce AppMaster.

Network Partition Characteristics		%
Partition any replica		44.9%
Partition a specific node		55.1%
▪ Partition the leader	36.0%	
▪ Partition a central service	8.8%	
▪ Partition a node with a special role	3.7%	
▪ Other (e.g., new node, source of data migration)	6.6%	

Finding 9. *The majority (88%) of the failures manifest by isolating a single node, with 45% of the failures manifest by isolating any replica*.

It is alarming that the majority of the failures can occur by isolating a single node. Conceivably, isolating a single node is more likely than other network-partitioning cases; it can happen because of an NIC failure, a single link failure, or a ToR switch failure. ToR switch failures are common in production networks leading to 40 network partitions in two years at Google [21] and 70% of the downtime at Microsoft [22]. This finding invalidates the common practice of assigning a low priority to ToR switch failures based on the presumption that data redundancy can effectively mask them [22]. Our results show that this practice aggravates the problem by prolonging the partition.

We further studied the connectivity between replicas (Table 10) of the same object and found that 45% of failures manifest by isolating any replica, and the rest requires isolating a specific node or service (e.g., ZooKeeper cluster). Among the failures that isolate a specific node, isolating a leader replica (36%) and central services (8.8%) are the most common. This does not reduce the possibility of a failure because, as in many systems, every node is a leader for some data and is a secondary replica for other data. Consequently, isolating any replica in the cluster will most likely isolate a leader.

This finding highlights the importance of testing these specific faults that isolate a leader, a central service, and nodes with special roles (e.g., scheduler, and MapReduce App Master).

5.2 Timing Constraints

Finding 10. The majority (*80%) of the failures are either deterministic or have known timing constraints*.

The majority of the failures (Table 11) are either deterministic (62%), meaning they will manifest given the input events, or have known timing constraints (18%). These known constraints are configurable or hard coded, such as the number of heartbeat periods to wait before declaring that a node has failed.

Table 11. Timing constraints.

Timing constraints		%
No timing constraints		61.8%
Has timing constraints		31.2%
▪ Known	18.4%	
▪ Unknown – but still can be tested	12.8%	
Nondeterministic		7%

Furthermore, we found that the timing constraints immediately follow network-partitioning faults. For instance, if a partition isolates a leader, for a failure to happen, events at the old leader side should be invoked right after the partition, so they are processed before the old leader steps down; while on the majority side, the test should sleep for a known period until a new leader is elected. For instance, in RabbitMQ, Redis, Hazelcast, and VoltDB, a failure will happen only if a write is issued before the old leader steps down (e.g., within three heartbeats) after a partitioning fault.

The 13% of the failures that have unknown timing constraints manifest when the sequence of events overlaps with a system internal operation. For instance, in Cassandra, a failure [85] will only occur if a network partition takes place during a data sync operation between the handoff node and a replica. However, these failures can still be tested by well-designed unit tests. For instance, to test the aforementioned Cassandra failure, a test should (1) isolate a replica to make the system add a handoff node. (2) Write a large amount of data. (3) Heal the partition. Now, the handoff node will start syncing the data with the replica. Finally, (4) create a network partition that isolates the replica during the sync operation and triggers the failure.

Only 7% of the failures are nondeterministic; these failures are caused by multithreaded interleavings and by overlapping the manifestation sequence with hard-to-predict internal system operations.

This finding implies that testers should pay close attention to timing. However, we identified that timing constraints usually follow the partitioning fault, which significantly simplifies testing.

5.3 Resolution Analysis

Finding 11. *The resolution of 47% of the failures required redesigning a system mechanism* (Table 12).

We consider a code patch to be fixing a design flaw if it involves significant changes to the affected mechanism logic, design, or protocol, such as implementing a new leader election protocol in MongoDB and changing configuration change protocols in Elasticsearch.

Table 12. Percentage of design and implementation flaws for failures reported in issue-tracking systems.

Category	%	Average Resolution Time
Design	46.6%	205 days
Implementation	32.2%	81 days
Unresolved	21.2%	-

The large percentage of the failures that led to changes in the mechanism design indicates that network-partitioning faults were not considered in the initial design phase. We expect that a design review focusing on network partitioning fault tolerance would have discovered systems vulnerability to these faults.

Table 12 also reports the resolution time, which is the period from the time a developer acknowledges a failure to the time the issue is fixed. Obtaining an accurate resolution time is tricky. We removed outliers that take minutes to commit a complex patch or take over two years to add a simple patch. In addition, it is not necessary for the time reported to be spent actively solving the issue. Nevertheless, because these are high-priority tickets, we think that the reported times give some indication of the resolution effort. Table 12 shows that design flaws take 2.5 times longer to resolve than implementation bugs.

We noticed that some systems opted to change the system specification instead of fixing the issue. For instance, Redis documentation states that "there is always a window of time when it is possible to lose writes during partitions" [86]. RabbitMQ's documentation was updated to indicate that locking does not tolerate network partitioning [87], and Hazelcast's documentation [88] states that it provides "best effort consistency," in which data updated through atomic operations may be lost. This could imply that some of the systems unnecessarily selected a strong consistency model where an eventual model was sufficient or the developers do not believe that these are high priority issues.

5.4 Opportunity for Improved Testing

Finding 12. *All failures can be reproduced on a cluster of five nodes, with the majority (83%) of the failures being reproducible with three nodes only* (Table 13).

This finding implies that it is not necessary to have a large cluster to test these systems. In fact, it is enough to test them using a single physical machine that runs five virtual machines.

Finding 13. *The majority of the failures (93%) can be reproduced through tests by using a fault injection framework such as NEAT.*

Considering our findings, perhaps it is not surprising that the majority of the failures can be reproduced using unit and system-level tests with a framework that can inject network-partitioning faults. The majority of the failures result from a single network-partitioning fault, need fewer than three common input events, and are

Table 13: Number of nodes needed to reproduce a failure.

Number of Nodes	%
3 nodes	83.1%
5 nodes	16.9%

deterministic or have bounded timing constraints. The 7% that cannot be easily tested are nondeterministic failures or have short vulnerability intervals.

6 Discussion

In this section, we address two additional observations:

- *Overlooking network-partitioning faults.* We found in many cases that the system designer did not consider the possibility of network partitioning. For example, Redis promises data reliability even though it uses asynchronous replication, leading to data loss [89]. Similarly, the Hazelcast locking service relies on asynchronous replication, leading to double locking [90]. Earlier versions of Aerospike assumed that the network is reliable [91].

  We found implicit assumptions made in the studied systems that are untrue. For instance, tickets from MapReduce, RabbitMQ, Ignite, and HBase indicate that the developer assumed an unreachable node to have halted, which is not true with network partitioning. Finally, all partial network-partitioning failures are caused by an implicit assumption that if a node can reach a service, then all nodes can reach that service, which is not always true.
- *Lack of adequate testing tools*. In general, we found that systems lack rigorous testing for network-partitioning. For unit tests related to the code patches we studied, the developers typically used mocking techniques to test the impact of network partitioning on one component on one side of the partition. This makes us believe that the community lacks a network-partitioning fault injection tool that can be integrated with the current testing frameworks.

7 NEAT Framework

We built the *ne*twork p*a*rtitioning *t*esting framework (NEAT), a testing framework with network-partitioning fault injection. NEAT supports the three types of partitions, has a simple API for creating and healing partitions, and simplifies the coordination of events across clients. NEAT is implemented in 1553 lines of Java and uses OpenFlow and the `iptables` tool to inject network-partitioning faults.

7.1 API

NEAT is a generic testing framework. It does not have any constraints on the target system. To test a system, the developer should implement three classes. First is the ISystem interface, which provides methods to install, start, obtain the status of, and shut down the target system. Second, is a Client class that provides wrappers around the client API (e.g., put or get calls). Third is the test workload and verification code.

Listing 1 presents a test for an Elasticsearch data loss failure [92] with partial network partitioning. The network partition (line 7) isolates s1 (the primary replica) and client 1 from s2 and client 2. However, all nodes can reach s3. s2 will detect that the primary replica (s1) is unreachable and start a leader election process. s3 will vote for s2, although it can reach s1, resulting in two leaders. Consequently, writes on both sides of the partition will succeed (line 11 and 12). After healing the partition (line 13), s2 will detect that s1 is reachable. As in Elasticsearch, the replica with a smaller ID wins the election, so s2 will step down and become a follower of s1. s2 will replicate s1's data and, consequently, all writes served by s2 during the partition will be lost and the check on line 16 will fail.

Listing 1. An Elasticsearch test for data loss. The system has three servers: s1 (primary node), s2, and s3, and two clients.

```
1  public static void testDataLoss(){
2   List<Node> side1 = asList(s1, client1);
3   // other servers and clients in one group
4   List<Node> side2 = asList(s2, client2);
5   // create a partial partition. s3 can reach
6   // all nodes
7   Partition netPart = Partitioner.partial(
8                             side1, side2);
9   sleep(SLEEP_LEADER_ELECTION_PERIOD);
10  // write to both sides of the partition
11  assertTrue(client1.write(obj1, v1));
12  assertTrue(client2.write(obj2, v2));
13  Partitioner.heal(netPart);
14  // verify the two objects
15  assertEquals(client2.read(obj1), v1);
16  assertEquals(client2.read(obj2), v2);    }
```

Listing 2. An ActiveMQ test for a double dequeue failure. The system has three servers and two clients.

```
1  public static void testDoubleDequeueu(){
2   assertTrue(client1.send(q1, msg1));
3   assertTrue(client1.send(q1, msg2));
4   // get the master node
5   Node master = AMQSys.getMaster(q1);
6   List<Node> minority= asList(master, client1);
7   List<Node>majority=Partitioner.rest(minority);
8   Partition netPart = Partitioner.complete(
9                             minority, majority);
10  // dequeue at both sides of the partition
11  Msg minMsg = client1.receive(q1);
12  sleep(SLEEP_PERIOD);
13  Msg majMsg = client2.receive(q1);
 assertNotEqual(minMsg, majMsg);     }
```

Listing 2 presents an ActiveMQ test for double dequeueing with complete network partitioning. The network partition (line 8) isolates the master and client1 from the rest of the cluster. The test then pops the queue at both sides of the partition (lines 11-13). If the two sides obtain the same value, then the value was dequeued twice and the test fails.

7.2 Creating and Healing Network Partitions

To create or heal a network partition, the developer calls one of the following methods.

- `Partition complete(List<Node> groupA, List<Node> groupB)`: creates a complete partition between groupA and groupB.

- `Partition partial(List<Node> groupA, List<Node> groupB)`: creates a partition between groupA and groupB without effecting their communication with the rest of the cluster.
- `Partition simplex(List<Node> groupSrc, List<Node> groupDst)`: creates a simplex partition such that packets can only flow from groupSrc to groupDst, but not in the other direction.
- `void heal(Partition p)`: heals partition p.

7.3 NEAT Design

NEAT has three components (Figure 2): server nodes, which run the target system; client nodes, which issue client requests; and a test engine. The test engine is a central node that runs the test workload (e.g., Listing 1).

The test engine simplifies testing by providing a global order for all client operations. The test engine invokes all client operations using Java RMI. The current NEAT prototype has two implementations of the network partitioner module: using OpenFlow and using the `iptables` tool. Furthermore, the test engine provides an API for crashing any group of nodes.

The OpenFlow-based partitioner is a network controller [35] that first installs the rules for a basic learning switch [93]. Then it installs partitioning rules to drop packets from a specific set of source IP addresses to a specific set of destination addresses. The partitioning rules are installed at a higher priority than the learning switch rules. The partitioner is implemented in 152 lines of code using Floodlight [94].

Our choice to use SDN to build a testing framework for distributed systems is research based. Connecting the nodes to a single switch and having the ability to monitor and control every packet in the system is a powerful capability for distributed systems testing. Our first attempt to explore this capability is to build a network partitioner for NEAT. Our current research effort explores techniques to collect detailed system traces under different failure scenarios and build tools to verify and visualize system protocols. This will help developers test, debug, and inspect protocols under different failure scenarios.

For deployments that do not have an OpenFlow switch, we implemented a partitioner by using the `iptables` tool to modify the firewall configuration on every node to create the specified partitions.

7.4 Testing Systems with NEAT

We used NEAT to test seven systems: Ceph [37] (v12.2.5), an object storage system; Apache Ignite [39] (v2.4.0), a key-value store and distributed data structures; Terracotta [40] (v4.3.4), a suite of distributed data structures; DKron [41] (v0.9.8), a job scheduling system; ActiveMQ [38] (v5.15.3), a message-queueing system; Infinispan [42] (v9.2.1), a key-value store; and MooseFS [43] (v3), a file system. All systems were configured with the most reliable configuration. For instance, when possible we persist data on disk, use synchronous replication, and set the minimum replication per operation to equal the majority or the number of all replicas.

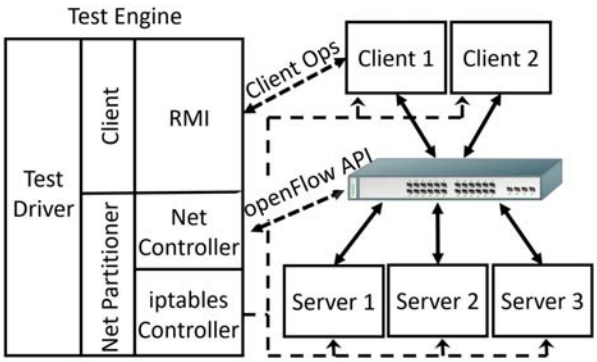


Figure 2. NEAT architecture.

Testing setup. We used two testbeds to run our experiments: CloudLab [95] and our own cluster. We used six nodes in our tests. The nodes were connected by a single switch. One node ran the test engine, three nodes ran the system, and two nodes acted as clients.

Our tests involved creating complete and partial partitions, then issuing simple client requests to the two sides of the partition, followed by performing a verification step. On average, tests are implemented in 30 lines of Java code.

The highlighted entries in Table 1 summarize our testing results. Our testing revealed 32 network-partitioning failures, out of which 30 are catastrophic. The failures we found lead to data loss, stale reads, data unavailability, double locking, and lock corruption. It is plausible that a single design flaw or implementation bug (e.g., flawed replication protocol) may cause failures in different operations (e.g., adding to a list and pushing to a queue). We count these as separate failures.

To demonstrate the versatility of NEAT, the following discusses failures that NEAT discovered.

Examples of complete network partition failures: We found that all Ignite atomic synchronization primitives, including semaphores, compare_and_set, increment_and_get, and decrement_and_get, are violated or corrupted when a complete network partition isolates one of the replicas. The main culprit of such failures is the assumption that an unreachable node has crashed; consequently, nodes on both sides of a partition remove the nodes they cannot reach (i.e., the nodes on the other side of the partition) from their replica set and continue to use the semaphore, which may lead to over counting the semaphore. Furthermore, an unreachable client that is holding a semaphore is assumed to have crashed. In this case, the system will reclaim the client's semaphore. If the partition heals

and the client signals the semaphore, the semaphore will be corrupted. These failures lead to lasting damage that persists after the partition heals.

Examples of partial network partition failures: ActiveMQ uses ZooKeeper to keep track of the current leader. If a partial network partition isolates the leader from the replicas, but not from ZooKeeper, the system will hang. The leader will not be able to forward messages to replicas and the replicas will not elect a new leader as ZooKeeper does not see the failure.

In DKron, if a partial partition separates the leader from the rest of DKron's nodes—but not from the central data store service—then the client requests processed by the leader will be successfully executed at the local level. However, DKron will indicate that the task has failed.

8 Additional Related Work

To the best of our knowledge, this is the first in-depth study of the manifestation sequence of network-partitioning failures. The manual analysis allowed us to examine the sequence of events in detail, identify common vulnerabilities, and find failure characteristics that can improve testing.

A large body of previous work analyzed failures in distributed systems. A subset of these efforts focused on specific component failures such as physical [96] and virtual machines [97], network devices [22, 24], storage systems [98, 99], software bugs [100], and job failures [101, 102, 103]. Another set characterized a broader set of failures, but only for specific domain of systems and services, such as HPC [104, 105, 106], IaaS clouds [107], data-mining services [108], hosting services [6, 109], and data-intensive systems [101, 100, 110]. Our work complements these efforts by focusing on failures triggered by network partitioning.

Yuan et al. [66] studied 198 randomly selected failures from six data analytics systems. Comparing our results, we find that a higher percentage of network-partitioning failures (80%) lead to catastrophic effects, compared to 24% reported by Yuan et al. [66]; and while 26% of general failures are nondeterministic, only 7% of network-partitioning failures are non-deterministic. These findings indicate that network-partitioning failures are more critical than general system failures, and testers need to pay close attention to timing.

Jepsen's blog posts report network-partitioning failures that were found using the Jepsen tool [54]. However, they do not detail the manifestation sequences, correlate failures across systems, study the impact of different types of network-partitioning faults, study client access requirements, characterize network faults, or analyze timing constraints.

Majumdar et al. [111] theoretically analyzed the space for faulty executions in the presence of complete network partitioning faults. They discussed the extreme size of the test space and the effectiveness of random testing if tests isolate a specific node, place a leader in a minority, and test with a random order of short sequences of operations.

While we identify characteristics to improve testing, our findings can inform other fault tolerance techniques. Previous efforts explored model checking [112, 113, 114, 115, 116], systematic fault injection [117, 118], and runtime verification techniques [119, 120] for improving systems' fault tolerance. Our findings inform these techniques to consider all types of network partitions and discovered characteristics that can improve these techniques' time and efficiency.

9 Insights

We conducted a comprehensive study of network-partitioning failures in modern cloud systems. It is surprising that these production systems experience silent catastrophic failures due to a frequently occurring infrastructure fault, when a single node is isolated, and under simple and common workloads. Our analysis identified that improvements to the software development process and testing can significantly improve systems' resilience to network partitions. These findings indicate that this is a high-impact research area that needs further effort to improve system design, engineering, testing, and fault tolerance. Our initial results with NEAT are encouraging; even our preliminary testing tool found bugs in production systems, indicating that there is a significant room for improvement.

Another interesting area of research that our analysis identified is partial network partitions fault tolerance. It is surprising that a large number of failures in production systems are triggered by this network fault, yet we could not find any discussion, failure model, or fault tolerance techniques that address this type of infrastructure fault.

Modern systems use unreachability as an indicator for node failure. Our analysis shows the dangers of this approach, as complete network partitions can isolate healthy nodes that lead to both sides assuming that the other side has crashed. Worse yet, partial partitions lead to a confusing state in which some nodes declare part of the system down while the rest of the nodes do not. Further, research is needed for building more accurate node-failure detectors and fault tolerance techniques.

While we identify better testing as one approach for improving system fault tolerance, we highlighted that the number of test cases one needs to consider is excessive. Luckily, our analysis found operations,

timing, ordering, and network failure characteristics that limit the testing space.

Our analysis highlights that the current network maintenance practice of assigning a low priority to ToR switch failure is ill founded and aggravates the problem. Finally, we highlight that system designers need to pay careful attention to internal and offline operations, need be wary of tweaking established protocols, and need to consider network partitioning failures early in their design process.

10 Conclusion and Future Work

We conducted an in-depth study of 136 failure reports from 25 widely used systems for failures triggered by network-partitioning faults. We present 13 main findings that can inform system designers, developers, testers, and administrators; and highlight the need for further research in network partitioning fault tolerance in general and with partial partitions in particular.

We built NEAT, a testing framework that can inject different types of network-partitioning faults. Our testing of seven systems revealed 32 failures.

In our current work, we are focusing on two directions: Extending NEAT to automate testing through workload and network fault generators and exploring fault tolerance techniques for partial network partitioning faults. Our data set and the source code are available at: https://dsl.uwaterloo.ca/projects/neat/

Acknowledgment

We thank the anonymous reviewers and our shepherd, Marcos Aguilera, for their insightful feedback. We thank Matei Ripeanu, Remzi Arpaci-Dusseau, Abdullah Gharaibeh, Ken Salem, Tim Brecht, and Bernard Wong for their insightful feedback on early versions of this paper. We thank Nicholas Lee, Alex Liu, Dian Tang, Anusan Sivakumaran, and Charles Wu for their help in reproducing some of the failures. This research was supported by an NSERC Discovery grant, NSERC Engage grant, Canada Foundation for Innovation (CFI) grant, and in-kind support from Google Canada.

References

[1] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: cluster computing with working sets," in *USENIX conference on Hot topics in cloud computing (HotCloud)*, 2010.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Symposium on operating systems design and implementation (OSDI)*, 2006.

[4] "Apache Hadoop," [Online]. Available: https://hadoop.apache.org/. [Accessed May 2018].

[5] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing,* vol. 5, no. 4, pp. 46-55, 2001 .

[6] D. Oppenheimer, A. Ganapathi and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, 2003.

[7] "Apache Hadoop 2.9.0 – HDFS High Availability," [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html. [Accessed 20 April 2018].

[8] "Linux-HA: Open Source High-Availability Software for Linux," [Online]. Available: http://www.linux-ha.org/wiki/Main_Page. [Accessed 27 April 2018].

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, 2008.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost and J. Furman, "Spanner: Google's Globally-Distributed Database," in *USENIX symposium on operating systems design and implementation (OSDI)*, Hollywood, CA, 2012.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding and J. Ferris, "TAO: Facebook's Distributed Data Store for the Social Graph," in *USENIX Annual Technical Conference (USENIX ATC)*, San Jose, CA, 2013.

[12] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett and H. V, "SPANStore: cost-effective geo-replicated storage spanning multiple cloud services," in *ACM symposium on operating systems principles (SOSP)*, Farminton, Pennsylvania, 2013.

[13] L. Lamport, "Paxos Made Simple," *ACM SIGACT News,* vol. 32, no. 4, pp. 18-25, 2001.

[14] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, Philadelphia, PA, 2014.

[15] T. D. Chandra, R. Griesemer and J. Redstone, "Paxos made live: an engineering perspective," in *ACM symposium on principles of distributed computing*, Portland, Oregon, USA, 2007.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value

store," in *Symposium on Operating systems principles (SOSP)*, Washington, USA, 2007.
[17] F. P. Junqueira, B. C. Reed and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, Hong Kong, 2011.
[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *ACM SIGOPS Operating Systems Review,* vol. 29, no. 5, pp. 172-182, 1995.
[19] D. B. Terry , V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
[20] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News,* vol. 33, no. 2, pp. 51-59, 2002.
[21] R. Govindan, I. Minei, M. Kallahalla , B. Koley and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," in *ACM SIGCOMM*, Florianopolis, Brazil, 2016.
[22] G. Phillipa , N. Jain and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM*, Toronto, 2011.
[23] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, A. C. Snoeren, D. Turner, K. Levchenko, J. C. Mogul, S. Savage and A. C. Snoeren, "On failure in managed enterprise networks," *Technical report HPL-2012-101, HP Labs,* 2012.
[24] D. Turner, K. Levchenko, A. C. Snoeren and S. Savage, "California fault lines: understanding the causes and impact of network failures," in *ACM SIGCOMM*, New York, NY, USA, 2010.
[25] "Apache Hadoop 2.6.0 - Fault Injection Framework and Development Guide," [Online]. Available: https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html. [Accessed April 2018].
[26] "Mockito framework," [Online]. Available: http://site.mockito.org/. [Accessed 27 April 2018].
[27] K. Beck, Test Driven Development: By Example., Addison-Wesley Professional, 2003.
[28] "Elasticsearch: RESTful, Distributed Search & Analytics," [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed October 2018].
[29] Hazelcast, "Hazelcast: the Leading In-Memory Data Grid," [Online]. Available: https://hazelcast.com/. [Accessed October 2018].
[30] J. Kreps, N. Narkhede and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," in *NetDB*, 2011.
[31] "MongoDB," [Online]. Available: https://www.mongodb.com/. [Accessed October 2018].
[32] "Redis: in-memory data structure store," [Online]. https://redis.io/. [Accessed October 2018].
[33] "VoltDB: In-Memory Database," [Online]. Available: https://www.voltdb.com/. [Accessed October 2018].
[34] A. Herr, "Veritas Cluster Server 6.2 I/O Fencing Deployment Considerations," Techincal report, Veritas Technologies, 2016.
[35] "OpenFlow Switch Specification, Version 1.5.1 (ONF TS-025)," Open Networking Foundation, 2015.
[36] "iptables: administration tool for IPv4 packet filtering and NAT," [Online]. Available: https://linux.die.net/man/8/iptables. [Accessed October 2018].
[37] "Ceph: distributed storage system," [Online]. Available: https://ceph.com/. [Accessed October 2018].
[38] "Apache ActiveMQ," [Online]. Available: http://activemq.apache.org/. [Accessed October 2018].
[39] "Ignite: Database and Caching Platform," [Online]. Available: https://ignite.apache.org/. [Accessed October 2018].
[40] "Terracotta data management platform," [Online]. Available: http://www.terracotta.org/. [Accessed October 2018].
[41] "Dkron: Distributed job scheduling system," [Online]. Available: https://dkron.io. [Accessed October 2018].
[42] "Infinispan: distributed in-memory key/value data store," [Online]. Available: http://infinispan.org/. [Accessed October 2018].
[43] "MooseFS: Moose file system," [Online]. Available: https://moosefs.com/. [Accessed October 2018].
[44] S. Jain, A. Kumar, M, S. al, J. Ong, L. Poutievski, A. Singh, S. Venkata, W, J. erer, J. Zhou and M. Zhu, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review,* 2013.
[45] "Data Center: Load Balancing Data Center, Solutions Reference Nework Design," Technical report, Cisco Systems, Inc., 2004.
[46] "bnx2 cards intermittantly going offline," [Online]. https://www.spinics.net/lists/netdev/msg210485.html. [Accessed October 2018].

[47] A. S. Tanenbaum and Maarten van Steen, Distributed Systems: Principles and Paradigms, 2nd ed., Pearson, 2006.

[48] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *VLDB Endowment,* vol. 1, no. 2, pp. 1277-1288, 2008.

[49] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards and V. Bedekar, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2011.

[50] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays," in *Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, 2004.

[51] "OpenStack Swift object storage," [Online]. Available: https://www.swiftstack.com/product/openstack-swift. [Accessed October 2018].

[52] "RethinkDB: the open-source database for the realtime web," [Online]. Available: https://www.rethinkdb.com/. [Accessed October 2018].

[53] "Protocol Buffers," [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed Octrober 2018].

[54] jepsen-io, "Jepsen: A framework for distributed systems verification, with fault injection," [Online]. Available: https://github.com/jepsen-io/jepsen. [Accessed October 2018].

[55] "Clojure programming language," [Online]. Available: https://clojure.org/. [Accessed October 2018].

[56] "Apache HBase," [Online]. Available: https://hbase.apache.org/. [Accessed October 2018].

[57] "Riak KV: distributed NoSQL key-value database," [Online]. Available: http://basho.com/riak/. [Accessed October 2018].

[58] "Apache Cassandra," [Online]. Available: http://cassandra.apache.org/. [Accessed October 2018].

[59] "Aerospike database 4," [Online]. Available: https://www.aerospike.com/. [Accessed October 2018].

[60] "Apache Geode data management solution," [Online]. Available: http://geode.apache.org/. [Accessed October 2018].

[61] "Apache ZooKeeper," [Online]. Available: https://zookeeper.apache.org/. [Accessed October 2018].

[62] "RabbitMQ message broker," [Online]. Available: https://www.rabbitmq.com/. [Accessed October 2018].

[63] "Chronos: Fault tolerant job scheduler for Mesos," [Online]. Available: https://mesos.github.io/chronos/. [Accessed October 2018].

[64] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *USENIX conference on Networked systems design and implementation (NSDI)*, Boston, MA, 2011.

[65] "Jepsen: MongoDB stale reads," [Online]. Available: https://aphyr.com/posts/322-jepsen-mongodb-stale-reads. [Accessed 17 March 2018].

[66] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain and M. Stumm, "Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems," in *USENIX conference on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, 2014.

[67] "Jepsen: Riak," [Online]. Available: https://aphyr.com/posts/285-call-me-maybe-riak. [Accessed 17 March 2018].

[68] "[SERVER-7008] socket exception [SEND_ERROR] on Mongo Sharding - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-7008. [Accessed 17 March 2018].

[69] "Network partition during peer discovery in auto clustering causes two clusters to form · Issue #1455 · rabbitmq/rabbitmq-server," [Online]. Available: https://github.com/rabbitmq/rabbitmq-server/issues/1455. [Accessed 17 March 2018].

[70] "[SERVER-17975] Stale reads with WriteConcern Majority and ReadPreference Primary - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-17975. [Accessed 17 March 2018].

[71] "[ENG-10389] Possible dirty read with RO transactions in certain partition scenarios - VoltDB JIRA," [Online]. Available: https://issues.voltdb.com/browse/ENG-10389. [Accessed 17 March 2018].

[72] "Possible write loss during cluster reconfiguration · Issue #5289 · rethinkdb/rethinkdb," [Online]. Available: https://github.com/rethinkdb/rethinkdb/issues/5289. [Accessed 17 March 2018].

[73] "[SERVER-14885] replica sets that disable chaining may have trouble electing a primary if

members have different priorities - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-14885. [Accessed 11 April 2018].
[74] "[ZOOKEEPER-2099] Using txnlog to sync a learner can corrupt the learner's datatree - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-2099. [Accessed 30 April 2018].
[75] "Disconnect between coordinating node and shards can cause duplicate updates or wrong status code · Issue #9967 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/9967. [Accessed 22 March 2018].
[76] "[HBASE-2312] Possible data loss when RS goes into GC pause while rolling HLog - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/HBASE-2312. [Accessed 1 May 2018].
[77] "[HDFS-577] Name node doesn't always properly recognize health of data node - ASF JIRA," [Online]. [Accessed 22 March 2018].
[78] "[MAPREDUCE-4819] AM can rerun job after reporting final job status to the client - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-4819. [Accessed 18 March 2018].
[79] "[HDFS-1384] NameNode should give client the first node in the pipeline from different rack other than that of excludedNodes list in the same rack. - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/HDFS-1384. [Accessed 17 March 2018].
[80] "[SERVER-27125] Arbiters in pv1 should vote no in elections if they can see a healthy primary of equal or greater priority to the candidate - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-27125. [Accessed 17 March 2018].
[81] "minimum_master_nodes does not prevent split-brain if splits are intersecting · Issue #2488 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/2488. [Accessed 17 March 2018].
[82] "Avoid Data Loss on Migration - Solution Design," [Online]. Available: https://hazelcast.atlassian.net/wiki/spaces/COM/pages/66519050/Avoid+Data+Loss+on+Migration+-+Solution+Design. [Accessed 28 March 2018].
[83] "PSYNC2 partial command backlog corruption · Issue #3899 · antirez/redis," [Online]. Available: https://github.com/antirez/redis/issues/3899. [Accessed 1 May 2018].
[84] "Deadlock while syncing mirrored queues · Issue #714 · rabbitmq/rabbitmq-server," [Online]. Available: https://github.com/rabbitmq/rabbitmq-server/issues/714. [Accessed 1 May 2018].
[85] "Cassandra removenode makes Gossiper Thread hang forever," [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-13562. [Accessed January 2018].
[86] "Redis Cluster Specification," [Online]. Available: https://redis.io/topics/cluster-spec. [Accessed 2018].
[87] "Distributed Semaphores with RabbitMQ," [Online]. Available: https://www.rabbitmq.com/blog/2014/02/19/distributed-semaphores-with-rabbitmq/. [Accessed 22 March 2018].
[88] "Consistency and Replication Model - Hazelcast Reference Manual," [Online]. Available: http://docs.hazelcast.org/docs/latest-development/manual/html/Consistency_and_Replication_Model.html. [Accessed 22 March 2018].
[89] "Jepsen: Redis," [Online]. Available: https://aphyr.com/posts/283-jepsen-redis. [Accessed 22 March 2018].
[90] "Jepsen: Hazelcast 3.8.3," [Online]. Available: https://jepsen.io/analyses/hazelcast-3-8-3. [Accessed 22 March 2018].
[91] "Jepsen: Aerospike," [Online]. Available: https://aphyr.com/posts/324-jepsen-aerospike. [Accessed 22 March 2018].
[92] "minimum_master_nodes does not prevent split-brain if splits are intersecting · Issue #2488 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/2488. [Accessed 17 May 2018].
[93] J. Kurose and K. Ross, Computer Networking: A Top-Down Approach, 7th ed., Pearson, 2016.
[94] "Floodlight OpenFlow Controller," [Online]. Available: http://www.projectfloodlight.org/floodlight/. [Accessed 19 March 2018].
[95] "CloudLab," [Online]. Available: https://www.cloudlab.us/. [Accessed May 2018].
[96] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *ACM symposium on Cloud computing (SoCC)*, New York, NY, USA, 2010.
[97] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann and T. Engbersen, "Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, USA, 2014.
[98] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes and S. Quinlna, "Availability in Globally Distributed Storage Systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, 2010.

[99] W. Jiang, C. Hu, Y. Zhou and A. Kanevsky, "Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics," *ACM Transactions on Storage ,* vol. 4, no. 3, p. Article 7, 2008.

[100] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin and A. D. Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *ACM Symposium on Cloud Computing (SOCC)*, New York, NY, USA, 2014.

[101] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin and T. Xie, "A Characteristic Study on Failures of Production Distributed Data-Parallel Programs," in *International Conference on Software Engineering (ICSE)*, 2013.

[102] X. Chen, C. D. Lu and K. Pattabiraman, "Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study," in *IEEE International Symposium on Software Reliability Engineering*, Naples, Italy, 2014.

[103] P. Garraghan, P. Townend and J. Xu, "An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment," in *IEEE International Symposium on High-Assurance Systems Engineering*, 2014.

[104] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Budapest, Hungary, 2013.

[105] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models," in *International Conference on Dependable Systems and Networks (DSN)*, Philadelphia, PA, USA, 2006.

[106] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing,* vol. 7, no. 4, pp. 337 - 350, 2010.

[107] T. Benson, S. Sahu, A. Akella and A. Shaikh, "A first look at problems in the cloud," in *USENIX conference on Hot topics in cloud computing (HotCloud)*, Boston, MA, 2010.

[108] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin and T. Qin, "An empirical study on quality issues of production big data platform," in *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[109] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama and K. J. Eliazar, "Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages," in *ACM Symposium on Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2016.

[110] A. Rabkin and R. H. Katz, "How Hadoop Clusters Break," *IEEE Software,* vol. 30, no. 4, pp. 88 - 94, 2012.

[111] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?," in *ACM Journal on Programming Languages*, 2017.

[112] P. Godefroid, "Model checking for programming languages using verisoft.," in *ACM symposium on principles of programming languages (POPL)*, Paris, 1997.

[113] S. Qadeer and D. Wu, "Kiss: keep it simple and sequential," in *Conf. on Programming Language Design and Implementation (PLDI)*, 2004.

[114] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman and H. S. Gunawi, "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[115] C. Baier and J.-P. Katoen, Principles of model checking, MIT press, 2008.

[116] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, 2009.

[117] P. Alvaro, J. Rosen and J. M. Hellerstein, "Lineage-driven Fault Injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[118] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[119] M. Pradel and T. R. Gross, "Automatic testing of sequential and concurrent substitutability," in *International Conference on Software Engineering (ICSE)*, 2013.

[120] T. Elmas, S. Tasiran and S. Qadeer, "VYRD: Verifying concurrent programs by runtime refinement-violation detection.," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang
Purdue University

Abstract

The monolithic server model where a server is the unit of deployment, operation, and failure is meeting its limits in the face of several recent hardware and application trends. To improve resource utilization, elasticity, heterogeneity, and failure handling in datacenters, we believe that datacenters should break monolithic servers into *disaggregated, network-attached hardware components*. Despite the promising benefits of hardware resource disaggregation, no existing OSes or software systems can properly manage it.

We propose a new OS model called the *splitkernel* to manage disaggregated systems. Splitkernel disseminates traditional OS functionalities into loosely-coupled *monitors*, each of which runs on and manages a hardware component. A splitkernel also performs resource allocation and failure handling of a distributed set of hardware components. Using the splitkernel model, we built *LegoOS*, a new OS designed for hardware resource disaggregation. LegoOS appears to users as a set of distributed servers. Internally, a user application can span multiple processor, memory, and storage hardware components. We implemented LegoOS on x86-64 and evaluated it by emulating hardware components using commodity servers. Our evaluation results show that LegoOS' performance is comparable to monolithic Linux servers, while largely improving resource packing and reducing failure rate over monolithic clusters.

1 Introduction

For many years, the unit of deployment, operation, and failure in datacenters has been a *monolithic server*, one that contains all the hardware resources that are needed to run a user program (typically a processor, some main memory, and a disk or an SSD). This monolithic architecture is meeting its limitations in the face of several issues and recent trends in datacenters.

First, datacenters face a difficult bin-packing problem of fitting applications to physical machines. Since a process can only use processor and memory in the same machine, it is hard to achieve full memory and CPU resource utilization [18, 33, 65]. Second, after packaging hardware devices in a server, it is difficult to add, remove, or change hardware components in datacenters [39]. Moreover, when a hardware component like a memory controller fails, the entire server is unusable. Finally, modern datacenters host increasingly heterogeneous hardware [5, 55, 84, 94]. However, designing new hardware that can fit into monolithic servers and deploying them in datacenters is a painful and cost-ineffective process that often limits the speed of new hardware adoption.

We believe that datacenters should break monolithic servers and organize hardware devices like CPU, DRAM, and disks as *independent, failure-isolated, network-attached components*, each having its own controller to manage its hardware. This *hardware resource disaggregation* architecture is enabled by recent advances in network technologies [24, 42, 52, 66, 81, 88] and the trend towards increasing processing power in hardware controller [9, 23, 92]. Hardware resource disaggregation greatly improves resource utilization, elasticity, heterogeneity, and failure isolation, since each hardware component can operate or fail on its own and its resource allocation is independent from other components. With these benefits, this new architecture has already attracted early attention from academia and industry [1, 15, 48, 56, 63, 77].

Hardware resource disaggregation completely shifts the paradigm of computing and presents a key challenge to system builders: *How to manage and virtualize the distributed, disaggregated hardware components?*

Unfortunately, existing kernel designs cannot address the new challenges hardware resource disaggregation brings, such as network communication overhead across disaggregated hardware components, fault tolerance of hardware components, and the resource management of distributed components. Monolithic kernels, microkernels [36], and exokernels [37] run one OS on a monolithic machine, and the OS assumes local accesses to shared main memory, storage devices, network interfaces, and other hardware resources in the machine. After disaggregating hardware resources, it may be viable to run the OS at a processor and remotely manage all other hardware components. However, remote management requires significant amount of network traffic, and when processors fail, other components are unusable. Multi-kernel OSes [21, 26, 76, 106] run a kernel at each processor (or core) in a monolithic computer and these per-processor kernels communicate with each other through message passing. Multi-kernels still assume local accesses to hardware resources in a monolithic machine and their message passing is over local buses instead of a general network. While existing OSes could be retrofitted to support hardware resource disaggregation, such retrofitting will be invasive to the central subsystems of an OS, such as memory and I/O management.

We propose *splitkernel*, a new OS architecture for

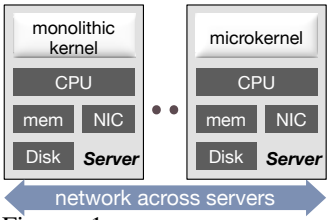


Figure 1: **OSes Designed for Monolithic Servers.**

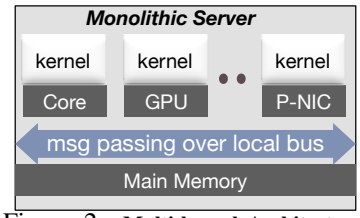


Figure 2: **Multi-kernel Architecture.** *P-NIC: programmable NIC.*

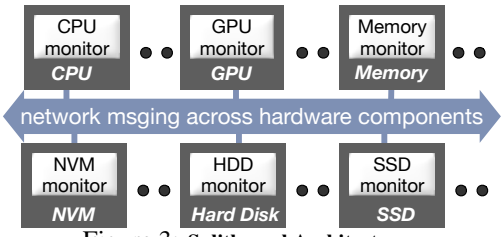


Figure 3: **Splitkernel Architecture.**

hardware resource disaggregation (Figure 3). The basic idea is simple: *When hardware is disaggregated, the OS should be also*. A splitkernel breaks traditional operating system functionalities into loosely-coupled *monitors*, each running at and managing a hardware component. Monitors in a splitkernel can be heterogeneous and can be added, removed, and restarted dynamically without affecting the rest of the system. Each splitkernel monitor operates locally for its own functionality and only communicates with other monitors when there is a need to access resources there. There are only two global tasks in a splitkernel: orchestrating resource allocation across components and handling component failure.

We choose not to support coherence across different components in a splitkernel. A splitkernel can use any general network to connect its hardware components. All monitors in a splitkernel communicate with each other via *network messaging* only. With our targeted scale, explicit message passing is much more efficient in network bandwidth consumption than the alternative of implicitly maintaining cross-component coherence.

Following the splitkernel model, we built LegoOS, the *first* OS designed for hardware resource disaggregation. LegoOS is a distributed OS that appears to applications as a set of virtual servers (called *vNodes*). A vNode can run on multiple processor, memory, and storage components and one component can host resources for multiple vNodes. LegoOS cleanly separates OS functionalities into three types of *monitors*, process monitor, memory monitor, and storage monitor. LegoOS monitors share no or minimal states and use a customized RDMA-based network stack to communicate with each other.

The biggest challenge and our focus in building LegoOS is the separation of processor and memory and their management. Modern processors and OSes assume all hardware memory units including main memory, page tables, and TLB are local. Simply moving all memory hardware and memory management software to across the network will not work.

Based on application properties and hardware trends, we propose a hardware plus software solution that cleanly separates processor and memory functionalities, while meeting application performance requirements. LegoOS moves all memory hardware units to the disaggregated memory components and organizes all levels of processor caches as virtual caches that are accessed using virtual memory addresses. To improve performance, LegoOS uses a small amount (*e.g.*, 4 GB) of DRAM organized as a virtual cache below current last-level cache.

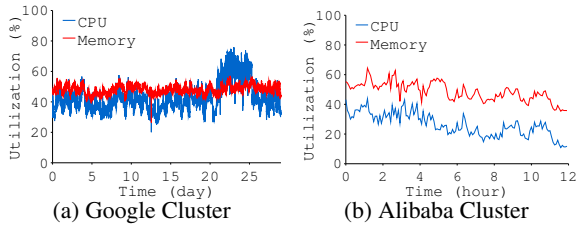
LegoOS process monitor manages application processes and the extended DRAM-cache. Memory monitor manages all virtual and physical memory space allocation and address mappings. LegoOS uses a novel two-level distributed virtual memory space management mechanism, which ensures efficient foreground memory accesses and balances load and space utilization at allocation time. Finally, LegoOS uses a space- and performance-efficient memory replication scheme to handle memory failure.

We implemented LegoOS on the x86-64 architecture. LegoOS is fully backward compatible with Linux ABIs by supporting common Linux system call APIs. To evaluate LegoOS, we emulate disaggregated hardware components using commodity servers. We evaluated LegoOS with microbenchmarks, the PARSEC benchmarks [22], and two unmodified datacenter applications, Phoenix [85] and TensorFlow [4]. Our evaluation results show that compared to monolithic Linux servers that can hold all the working sets of these applications, LegoOS is only 1.3× to 1.7× slower with 25% of application working set available as DRAM cache at processor components. Compared to monolithic Linux servers whose main memory size is the same as LegoOS' DRAM cache size and which use local SSD/DRAM swapping or network swapping, LegoOS' performance is 0.8× to 3.2×. At the same time, LegoOS largely improves resource packing and reduces system mean time to failure.

Overall, this work makes the following contributions:

- We propose the concept of splitkernel, a new OS architecture that fits the hardware resource disaggregation architecture.
- We built LegoOS, the first OS that runs on and manages a disaggregated hardware cluster.
- We propose a new hardware architecture to cleanly separate processor and memory hardware functionalities, while preserving most of the performance of monolithic server architecture.

LegoOS is publicly available at *http://LegoOS.io*.



(a) Google Cluster (b) Alibaba Cluster

Figure 4: **Datacenter Resource Utilization.**

2 Disaggregate Hardware Resource

This section motivates the hardware resource disaggregation architecture and discusses the challenges in managing disaggregated hardware.

2.1 Limitations of Monolithic Servers

A monolithic server has been the unit of deployment and operation in datacenters for decades. This long-standing *server-centric* architecture has several key limitations.

Inefficient resource utilization. With a server being the physical boundary of resource allocation, it is difficult to fully utilize all resources in a datacenter [18, 33, 65]. We analyzed two production cluster traces: a 29-day Google one [45] and a 12-hour Alibaba one [10]. Figure 4 plots the aggregated CPU and memory utilization in the two clusters. For both clusters, only around half of the CPU and memory are utilized. Interestingly, a significant amount of jobs are being evicted at the same time in these traces (*e.g.*, evicting low-priority jobs to make room for high-priority ones [102]). One of the main reasons for resource underutilization in these production clusters is the constraint that CPU and memory for a job have to be allocated from the same physical machine.

Poor hardware elasticity. It is difficult to add, move, remove, or reconfigure hardware components after they have been installed in a monolithic server [39]. Because of this rigidity, datacenter owners have to plan out server configurations in advance. However, with today's speed of change in application requirements, such plans have to be adjusted frequently, and when changes happen, it often comes with waste in existing server hardware.

Coarse failure domain. The failure unit of monolithic servers is coarse. When a hardware component within a server fails, the whole server is often unusable and applications running on it can all crash. Previous analysis [90] found that motherboard, memory, CPU, power supply failures account for 50% to 82% of hardware failures in a server. Unfortunately, monolithic servers cannot continue to operate when any of these devices fail.

Bad support for heterogeneity. Driven by application needs, new hardware technologies are finding their ways into modern datacenters [94]. Datacenters no longer host only commodity servers with CPU, DRAM, and hard disks. They include non-traditional and specialized hardware like GPGPU [11, 46], TPU [55], DPU [5], FPGA [12, 84], non-volatile memory [49], and NVMe-based SSDs [98]. The monolithic server model tightly couples hardware devices with each other and with a motherboard. As a result, making new hardware devices work with existing servers is a painful and lengthy process [84]. Mover, datacenters often need to purchase new servers to host certain hardware. Other parts of the new servers can go underutilized and old servers need to retire to make room for new ones.

2.2 Hardware Resource Disaggregation

The server-centric architecture is a bad fit for the fast-changing datacenter hardware, software, and cost needs. There is an emerging interest in utilizing resources beyond a local machine [41], such as distributed memory [7, 34, 74, 79] and network swapping [47]. These solutions improve resource utilization over traditional systems. However, they cannot solve all the issues of monolithic servers (*e.g.*, the last three issues in §2.1), since their hardware model is still a monolithic one. To fully support the growing heterogeneity in hardware and to provide elasticity and flexibility at the hardware level, we should *break the monolithic server model.*

We envision a *hardware resource disaggregation* architecture where hardware resources in traditional servers are disseminated into network-attached *hardware components*. Each component has a controller and a network interface, can operate on its own, and is an *independent, failure-isolated* entity.

The disaggregated approach largely increases the flexibility of a datacenter. Applications can freely use resources from any hardware component, which makes resource allocation easy and efficient. Different types of hardware resources can *scale independently*. It is easy to add, remove, or reconfigure components. New types of hardware components can easily be deployed in a datacenter — by simply enabling the hardware to talk to the network and adding a new network link to connect it. Finally, hardware resource disaggregation enables fine-grain failure isolation, since one component failure will not affect the rest of a cluster.

Three hardware trends are making resource disaggregation feasible in datacenters. First, network speed has grown by more than an order of magnitude and has become more scalable in the past decade with new technologies like Remote Direct Memory Access (*RDMA*) [69] and new topologies and switches [15, 30, 31], enabling fast accesses of hardware components that are disaggregated across the network. InfiniBand will soon reach 200Gbps and sub-600 nanosecond speed [66], being only 2× to 4× slower than main memory bus in bandwidth. With main memory bus facing a bandwidth wall [87], future network bandwidth (at line rate) is even projected to exceed local DRAM bandwidth [99].

Second, network interfaces are moving closer to hardware components, with technologies like Intel Omni-Path [50], RDMA [69], and NVMe over Fabrics [29, 71]. As a result, hardware devices will be able to access network directly without the need to attach any processors.

Finally, hardware devices are incorporating more processing power [8, 9, 23, 67, 68, 75], allowing application and OS logics to be offloaded to hardware [57, 92]. On-device processing power will enable system software to manage disaggregated hardware components locally.

With these hardware trends and the limitations of monolithic servers, we believe that future datacenters will be able to largely benefit from hardware resource disaggregation. In fact, there have already been several initial hardware proposals in resource disaggregation [1], including disaggregated memory [63, 77, 78], disaggregated flash [59, 60], Intel Rack-Scale System [51], HP "The Machine" [40, 48], IBM Composable System [28], and Berkeley Firebox [15].

2.3 OSes for Resource Disaggregation

Despite various benefits hardware resource disaggregation promises, it is still unclear how to manage or utilize disaggregated hardware in a datacenter. Unfortunately, existing OSes and distributed systems cannot work well with this new architecture. Single-node OSes like Linux view a server as the unit of management and assume all hardware components are local (Figure 1). A potential approach is to run these OSes on processors and access memory, storage, and other hardware resources remotely. Recent disaggregated systems like soNUMA [78] take this approach. However, this approach incurs high network latency and bandwidth consumption with remote device management, misses the opportunity of exploiting device-local computation power, and makes processors the single point of failure.

Multi-kernel solutions [21, 26, 76, 106, 107] (Figure 2) view different cores, processors, or programmable devices within a server separately by running a kernel on each core/device and using message passing to communicate across kernels. These kernels still run in a single server and all access some common hardware resources in the server like memory and the network interface. Moreover, they do not manage distributed resources or handle failures in a disaggregated cluster.

There have been various distributed OS proposals, most of which date decades back [16, 82, 97]. Most of these distributed OSes manage a set of monolithic servers instead of hardware components.

Hardware resource disaggregation is fundamentally different from the traditional monolithic server model. A complete disaggregation of processor, memory, and storage means that when managing one of them, there will be no local accesses to the other two. For example, processors will have no local memory or storage to store user or kernel data. An OS also needs to manage distributed hardware resource and handle hardware component failure. We summarize the following key challenges in building an OS for resource disaggregation, some of which have previously been identified [40].

- How to deliver good performance when application execution involves the access of network-partitioned disaggregated hardware and current network is still slower than local buses?
- How to locally manage individual hardware components with limited hardware resources?
- How to manage distributed hardware resources?
- How to handle a component failure without affecting other components or running applications?
- What abstraction should be exposed to users and how to support existing datacenter applications?

Instead of retrofitting existing OSes to confront these challenges, we take the approach of designing a new OS architecture from the ground up for hardware resource disaggregation.

3 The Splitkernel OS Architecture

We propose *splitkernel*, a new OS architecture for resource disaggregation. Figure 3 illustrates splitkernel's overall architecture. The splitkernel disseminates an OS into pieces of different functionalities, each running at and managing a hardware component. All components communicate by message passing over a common network, and splitkernel globally manages resources and component failures. Splitkernel is a general OS architecture we propose for hardware resource disaggregation. There can be many types of implementation of splitkernel. Further, we make no assumption on the specific hardware or network type in a disaggregated cluster a splitkernel runs on. Below, we describe four key concepts of the splitkernel architecture.

Split OS functionalities. Splitkernel breaks traditional OS functionalities into *monitors*. Each monitor manages a hardware component, virtualizes and protects its physical resources. Monitors in a splitkernel are loosely-coupled and they communicate with other monitors to access remote resources. For each monitor to operate on its own with minimal dependence on other monitors, we use a stateless design by sharing no or minimal *states*, or metadata, across monitors.

Run monitors at hardware components. We expect each non-processor hardware component in a disaggregated cluster to have a controller that can run a monitor. A hardware controller can be a low-power general-purpose core, an ASIC, or an FPGA. Each monitor in a splitkernel can use its own implementation to manage the hardware

component it runs on. This design makes it easy to integrate heterogeneous hardware in datacenters — to deploy a new hardware device, its developers only need to build the device, implement a monitor to manage it, and attach the device to the network. Similarly, it is easy to reconfigure, restart, and remove hardware components.

Message passing across non-coherent components. Unlike other proposals of disaggregated systems [48] that rely on coherent interconnects [24, 42, 81], a splitkernel runs on general-purpose network layer like Ethernet and neither underlying hardware nor the splitkernel provides cache coherence across components. We made this design choice mainly because maintaining coherence for our targeted cluster scale would cause high network bandwidth consumption. Instead, all communication across components in a splitkernel is through *network messaging*. A splitkernel still retains the coherence guarantee that hardware already provides within a component (*e.g.*, cache coherence across cores in a CPU), and applications running on top of a splitkernel can use message passing to implement their desired level of coherence for their data across components.

Global resource management and failure handling. One hardware component can host resources for multiple applications and its failure can affect all these applications. In addition to managing individual components, the splitkernel also needs to globally manage resources and failure. To minimize performance and scalability bottleneck, the splitkernel only involves global resource management occasionally for coarse-grained decisions, while individual monitors make their own fine-grained decisions. The splitkernel handles component failure by adding redundancy for recovery.

4 LegoOS Design

Based on the splitkernel architecture, we built *LegoOS*, the first OS designed for hardware resource disaggregation. LegoOS is a research prototype that demonstrates the feasibility of the splitkernel design, but it is not the only way to build a splitkernel. LegoOS' design targets three types of hardware components: processor, memory, and storage, and we call them *pComponent, mComponent*, and *sComponent*.

This section first introduces the abstraction LegoOS exposes to users and then describes the hardware architecture of components LegoOS runs on. Next, we explain the design of LegoOS' process, memory, and storage monitors. Finally, we discuss LegoOS' global resource management and failure handling mechanisms.

Overall, LegoOS achieves the following design goals:

- Clean separation of process, memory, and storage functionalities.
- Monitors run at hardware components and fit device constraints.
- Comparable performance to monolithic Linux servers.
- Efficient resource management and memory failure handling, both in space and in performance.
- Easy-to-use, backward compatible user interface.
- Supports common Linux system call interfaces.

4.1 Abstraction and Usage Model

LegoOS exposes a distributed set of *virtual nodes*, or *vNode*, to users. From users' point of view, a vNode is like a virtual machine. Multiple users can run in a vNode and each user can run multiple processes. Each vNode has a unique ID, a unique virtual IP address, and its own storage mount point. LegoOS protects and isolates the resources given to each vNode from others. Internally, one vNode can run on multiple pComponents, multiple mComponents, and multiple sComponents. At the same time, each hardware component can host resources for more than one vNode. The internal execution status is transparent to LegoOS users; they do not know which physical components their applications run on.

With splitkernel's design principle of components not being coherent, LegoOS does not support writable shared memory across processors. LegoOS assumes that threads within the same process access shared memory and threads belonging to different processes do not share writable memory, and LegoOS makes scheduling decision based on this assumption (§4.3.1). Applications that use shared writable memory across processes (*e.g.*, with MAP_SHARED) will need to be adapted to use message passing across processes. We made this decision because writable shared memory across processes is rare (we have not seen a single instance in the datacenter applications we studied), and supporting it makes both hardware and software more complex (in fact, we have implemented this support but later decided not to include it because of its complexity).

One of the initial decisions we made when building LegoOS is to support the Linux system call interface and unmodified Linux ABI, because doing so can greatly ease the adoption of LegoOS. Distributed applications that run on Linux can seamlessly run on a LegoOS cluster by running on a set of vNodes.

4.2 Hardware Architecture

LegoOS pComponent, mComponent, and sComponent are independent devices, each having their own hardware controller and network interface (for pComponent, the hardware controller is the processor itself). Our current hardware model uses CPU in pComponent, DRAM in mComponent, and SSD or HDD in sComponent. We leave exploring other hardware devices for future work.

To demonstrate the feasibility of hardware resource disaggregation, we propose a pComponent and an

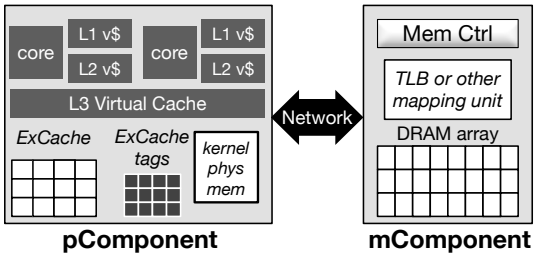


Figure 5: **LegoOS pComponent and mComponent Architecture.**

mComponent architecture designed within today's network, processor, and memory performance and hardware constraints (Figure 5).

Separating process and memory functionalities. LegoOS moves all hardware memory functionalities to mComponents (e.g., page tables, TLBs) and leaves *only* caches at the pComponent side. With a clean separation of process and memory hardware units, the allocation and management of memory can be completely transparent to pComponents. Each mComponent can choose its own memory allocation technique and virtual to physical memory address mappings (*e.g.*, segmentation).

Processor virtual caches. After moving all memory functionalities to mComponents, pComponents will only see virtual addresses and have to use virtual memory addresses to access its caches. Because of this, LegoOS organizes all levels of pComponent caches as *virtual caches* [44, 104], *i.e.*, virtually-indexed and virtually-tagged caches.

A virtual cache has two potential problems, commonly known as synonyms and homonyms [95]. Synonyms happens when a physical address maps to multiple virtual addresses (and thus multiple virtual cache lines) as a result of memory sharing across processes, and the update of one virtual cache line will not reflect to other lines that share the data. Since LegoOS does not allow writable inter-process memory sharing, it will not have the synonym problem. The homonym problem happens when two address spaces use the same virtual address for their own different data. Similar to previous solutions [20], we solve homonyms by storing an address space ID (ASID) with each cache line, and differentiate a virtual address in different address spaces using ASIDs.

Separating memory for performance and for capacity. Previous studies [41, 47] and our own show that today's network speed cannot meet application performance requirements if all memory accesses are across the network. Fortunately, many modern datacenter applications exhibit strong memory access temporal locality. For example, we found 90% of memory accesses in PowerGraph [43] go to just 0.06% of total memory and 95% go to 3.1% of memory (22% and 36% for TensorFlow [4] respectively, 5.1% and 6.6% for Phoenix [85]).

With good memory-access locality, we propose to leave a small amount of memory (*e.g.*, 4 GB) at each pComponent and move most memory across the network (*e.g.*, few TBs per mComponent). pComponents' local memory can be regular DRAM or the on-die HBM [53, 72], and mComponents use DRAM or NVM.

Different from previous proposals [63], we propose to organize pComponents' DRAM/HBM as cache rather than main memory for a clean separation of process and memory functionalities. We place this cache under the current processor Last-Level Cache (LLC) and call it an extended cache, or *ExCache*. ExCache serves as another layer in the memory hierarchy between LLC and memory across the network. With this design, ExCache can serve hot memory accesses fast, while mComponents can provide the capacity applications desire.

ExCache is a virtual, inclusive cache, and we use a combination of hardware and software to manage ExCache. Each ExCache line has a (virtual-address) tag and two access permission bits (one for read/write and one for valid). These bits are set by software when a line is inserted to ExCache and checked by hardware at access time. For best hit performance, the hit path of ExCache is handled purely by hardware — the hardware cache controller maps a virtual address to an ExCache set, fetches and compares tags in the set, and on a hit, fetches the hit ExCache line. Handling misses of ExCache is more complex than with traditional CPU caches, and thus we use LegoOS to handle the miss path of ExCache (see §4.3.2).

Finally, we use a small amount of DRAM/HBM at pComponent for LegoOS' own kernel data usages, accessed directly with physical memory addresses and managed by LegoOS. LegoOS ensures that all its own data fits in this space to avoid going to mComponents.

With our design, pComponents do not need any address mappings: LegoOS accesses all pComponent-side DRAM/HBM using physical memory addresses and does simple calculations to locate the ExCache set for a memory access. Another benefit of not handling address mapping at pComponents and moving TLBs to mComponents is that pComponents do not need to access TLB or suffer from TLB misses, potentially making pComponent cache accesses faster [58].

4.3 Process Management

The LegoOS *process monitor* runs in the kernel space of a pComponent and manages the pComponent's CPU cores and ExCache. pComponents run user programs in the user space.

4.3.1 Process Management and Scheduling

At every pComponent, LegoOS uses a simple local thread scheduling model that targets datacenter applications (we will discuss global scheduling in § 4.6). LegoOS dedicates a small amount of cores for kernel back-

ground threads (currently two to four) and uses the rest of the cores for application threads. When a new process starts, LegoOS uses a global policy to choose a pComponent for it (§ 4.6). Afterwards, LegoOS schedules new threads the process spawns on the same pComponent by choosing the cores that host fewest threads. After assigning a thread to a core, we let it run to the end with no scheduling or kernel preemption under common scenarios. For example, we do not use any network interrupts and let threads busy wait on the completion of outstanding network requests, since a network request in LegoOS is fast (*e.g.*, fetching an ExCache line from an mComponent takes around 6.5 μs). LegoOS improves the overall processor utilization in a disaggregated cluster, since it can freely schedule processes on any pComponents without considering memory allocation. Thus, we do not push for perfect core utilization when scheduling individual threads and instead aim to minimize scheduling and context switch performance overheads. Only when a pComponent has to schedule more threads than its cores will LegoOS start preempting threads on a core.

4.3.2 ExCache Management

LegoOS process monitor configures and manages ExCache. During the pComponent's boot time, LegoOS configures the set associativity of ExCache and its cache replacement policy. While ExCache hit is handled completely in hardware, LegoOS handles misses in software. When an ExCache miss happens, the process monitor fetches the corresponding line from an mComponent and inserts it to ExCache. If the ExCache set is full, the process monitor first evicts a line in the set. It throws away the evicted line if it is clean and writes it back to an mComponent if it is dirty. LegoOS currently supports two eviction policies: FIFO and LRU. For each ExCache set, LegoOS maintains a FIFO queue (or an approximate LRU list) and chooses ExCache lines to evict based on the corresponding policy (see §5.3 for details).

4.3.3 Supporting Linux Syscall Interface

One of our early decisions is to support Linux ABIs for backward compatibility and easy adoption of LegoOS. A challenge in supporting the Linux system call interface is that many Linux syscalls are associated with *states*, information about different Linux subsystems that is stored with each process and can be accessed by user programs across syscalls. For example, Linux records the states of a running process' open files, socket connections, and several other entities, and it associates these states with file descriptors (*fd*s) that are exposed to users. In contrast, LegoOS aims at the clean separation of OS functionalities. With LegoOS' stateless design principle, each component only stores information about its own resource and each request across components contains all the information that the destination component needs to handle the request. To solve this discrepancy between the Linux syscall interface and LegoOS' design, we add a layer on top of LegoOS' core process monitor at each pComponent to store Linux states and translate these states and the Linux syscall interface to LegoOS' internal interface.

4.4 Memory Management

We use mComponents for three types of data: anonymous memory (*i.e.*, heaps, stacks), memory-mapped files, and storage buffer caches. The LegoOS *memory monitor* manages both the virtual and physical memory address spaces, their allocation, deallocation, and memory address mappings. It also performs the actual memory read and write. No user processes run on mComponents and they run completely in the kernel mode (same is true for sComponents).

LegoOS lets a process address space span multiple mComponents to achieve efficient memory space utilization and high parallelism. Each application process uses one or more mComponents to host its data and a *home mComponent*, an mComponent that initially loads the process, accepts and oversees all system calls related to virtual memory space management (*e.g.*, `brk`, `mmap`, `munmap`, and `mremap`). LegoOS uses a global memory resource manager (*GMM*) to assign a home mComponent to each new process at its creation time. A home mComponent can also host process data.

4.4.1 Memory Space Management

Virtual memory space management. We propose a two-level approach to manage distributed virtual memory spaces, where the home mComponent of a process makes coarse-grained, high-level virtual memory allocation decisions and other mComponents perform fine-grained virtual memory allocation. This approach minimizes network communication during both normal memory accesses and virtual memory operations, while ensuring good load balancing and memory utilization. Figure 6 demonstrates the data structures used.

At the higher level, we split each virtual memory address space into coarse-grained, fix-sized *virtual regions*, or *vRegions* (*e.g.*, of 1 GB). Each vRegion that contains allocated virtual memory addresses (an active vRegion) is *owned* by an mComponent. The owner of a vRegion handles all memory accesses and virtual memory requests within the vRegion.

The lower level stores user process virtual memory area (*vma*) information, such as virtual address ranges and permissions, in *vma trees*. The owner of an active vRegion stores a vma tree for the vRegion, with each node in the tree being one vma. A user-perceived virtual memory range can split across multiple mComponents, but only one mComponent owns a vRegion.

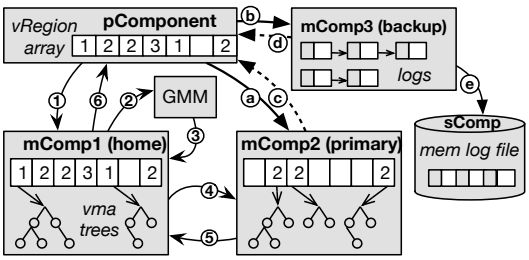


Figure 6: **Distributed Memory Management.**

vRegion owners perform the actual virtual memory allocation and vma tree set up. A home mComponent can also be the owner of vRegions, but the home mComponent does not maintain any information about memory that belongs to vRegions owned by other mComponents. It only keeps the information of which mComponent owns a vRegion (in a *vRegion array*) and how much free virtual memory space is left in each vRegion. These metadata can be easily reconstructed if a home mComponent fails.

When an application process wants to allocate a virtual memory space, the pComponent forwards the allocation request to its home mComponent (① in Figure 6). The home mComponent uses its stored information of available virtual memory space in vRegions to find one or more vRegions that best fit the requested amount of virtual memory space. If no active vRegion can fit the allocation request, the home mComponent makes a new vRegion active and contacts the GMM (② and ③) to find a candidate mComponent to own the new vRegion. GMM makes this decision based on available physical memory space and access load on different mComponents (§ 4.6). If the candidate mComponent is not the home mComponent, the home mComponent next forwards the request to that mComponent (④), which then performs local virtual memory area allocation and sets up the proper vma tree. Afterwards, the pComponent directly sends memory access requests to the owner of the vRegion where the memory access falls into (*e.g.*, ⓐ and ⓒ in Figure 6).

LegoOS' mechanism of distributed virtual memory management is efficient and it cleanly separates memory operations from pComponents. pComponents hand over all memory-related system call requests to mComponents and only cache a copy of the vRegion array for fast memory accesses. To fill a cache miss or to flush a dirty cache line, a pComponent looks up the cached vRegion array to find its owner mComponent and sends the request to it.

Physical memory space management. Each mComponent manages the physical memory allocation for data that falls into the vRegion that it owns. Each mComponent can choose their own way of physical memory allocation and own mechanism of virtual-to-physical memory address mapping.

4.4.2 Optimization on Memory Accesses

With our strawman memory management design, all ExCache misses will go to mComponents. We soon found that a large performance overhead in running real applications is caused by filling empty ExCache, *i.e.*, *cold misses*. To reduce the performance overhead of cold misses, we propose a technique to avoid accessing mComponent on first memory accesses.

The basic idea is simple: since the initial content of anonymous memory (non-file-backed memory) is zero, LegoOS can directly allocate a cache line with empty content in ExCache for the first access to anonymous memory instead of going to mComponent (we call such cache lines *p-local lines*). When an application creates a new anonymous memory region, the process monitor records its address range and permission. The application's first access to this region will be an ExCache miss and it will trap to LegoOS. LegoOS process monitor then allocates an ExCache line, fills it with zeros, and sets its R/W bit according to the recorded memory region's permission. Before this p-local line is evicted, it only lives in the ExCache. No mComponents are aware of it or will allocate physical memory or a virtual-to-physical memory mapping for it. When a p-local cache line becomes dirty and needs to be flushed, the process monitor sends it to its owner mComponent, which then allocates physical memory space and establishes a virtual-to-physical memory mapping. Essentially, LegoOS *delays physical memory allocation until write time*. Notice that it is safe to only maintain p-local lines at a pComponent ExCache without any other pComponents knowing them, since pComponents in LegoOS do not share any memory and other pComponents will not access this data.

4.5 Storage Management

LegoOS supports a hierarchical file interface that is backward compatible with POSIX through its vNode abstraction. Users can store their directories and files under their vNodes' mount points and perform normal read, write, and other accesses to them.

LegoOS implements core storage functionalities at sComponents. To cleanly separate storage functionalities, LegoOS uses a stateless storage server design, where each I/O request to the storage server contains all the information needed to fulfill this request, *e.g.*, full path name, absolute file offset, similar to the server design in NFS v2 [89].

While LegoOS supports a hierarchical file use interface, internally, LegoOS storage monitor treats (full) directory and file paths just as unique names of a file and place all files of a vNode under one internal directory at the sComponent. To locate a file, LegoOS storage monitor maintains a simple hash table with the full paths of files (and directories) as keys. From our observation,

most datacenter applications only have a few hundred files or less. Thus, a simple hash table for a whole vNode is sufficient to achieve good lookup performance. Using a non-hierarchical file system implementation largely reduces the complexity of LegoOS' file system, making it possible for a storage monitor to fit in storage devices controllers that have limited processing power [92].

LegoOS places the storage buffer cache at mComponents rather than at sComponents, because sComponents can only host a limited amount of internal memory. LegoOS memory monitor manages the storage buffer cache by simply performing insertion, lookup, and deletion of buffer cache entries. For simplicity and to avoid coherence traffic, we currently place the buffer cache of one file under one mComponent. When receiving a file read system call, the LegoOS process monitor first uses its extended Linux state layer to look up the full path name, then passes it with the requested offset and size to the mComponent that holds the file's buffer cache. This mComponent will look up the buffer cache and returns the data to pComponent on a hit. On a miss, mComponent will forward the request to the sComponent that stores the file, which will fetch the data from storage device and return it to the mComponent. The mComponent will then insert it into the buffer cache and returns it to the pComponent. Write and fsync requests work in a similar fashion.

4.6 Global Resource Management

LegoOS uses a two-level resource management mechanism. At the higher level, LegoOS uses three global resource managers for process, memory, and storage resources, *GPM, GMM*, and *GSM*. These global managers perform coarse-grained global resource allocation and load balancing, and they can run on one normal Linux machine. Global managers only maintain approximate resource usage and load information. They update their information either when they make allocation decisions or by periodically asking monitors in the cluster. At the lower level, each monitor can employ its own policies and mechanisms to manage its local resources.

For example, process monitors allocate new threads locally and only ask GPM when they need to create a new process. GPM chooses the pComponent that has the least amount of threads based on its maintained approximate information. Memory monitors allocate virtual and physical memory space on their own. Only home mComponent asks GMM when it needs to allocate a new vRegion. GMM maintains approximate physical memory space usages and memory access load by periodically asking mComponents and chooses the memory with least load among all the ones that have at least vRegion size of free physical memory.

LegoOS decouples the allocation of different resources and can freely allocate each type of resource from a pool of components. Doing so largely improves resource packing compared to a monolithic server cluster that packs all type of resources a job requires within one physical machine. Also note that LegoOS allocates hardware resources only *on demand*, *i.e.*, when applications actually create threads or access physical memory. This on-demand allocation strategy further improves LegoOS' resource packing efficiency and allows more aggressive over-subscription in a cluster.

4.7 Reliability and Failure Handling

After disaggregation, pComponents, mComponents, and sComponents can all fail independently. Our goal is to build a reliable disaggregated cluster that has the same or lower application failure rate than a monolithic cluster. As a first (and important) step towards achieving this goal, we focus on providing memory reliability by handling mComponent failure in the current version of LegoOS because of three observations. First, when distributing an application's memory to multiple mComponents, the probability of memory failure increases and not handling mComponent failure will cause applications to fail more often on a disaggregated cluster than on monolithic servers. Second, since most modern datacenter applications already provide reliability to their distributed storage data and the current version of LegoOS does not split a file across sComponent, we leave providing storage reliability to applications. Finally, since LegoOS does not split a process across pComponents, the chance of a running application process being affected by the failure of a pComponent is similar to one affected by the failure of a processor in a monolithic server. Thus, we currently do not deal with pComponent failure and leave it for future work.

A naive approach to handle memory failure is to perform a full replication of memory content over two or more mComponents. This method would require at least $2\times$ memory space, making the monetary and energy cost of providing reliability prohibitively high (the same reason why RAMCloud [80] does not replicate in memory). Instead, we propose a space- and performance-efficient approach to provide in-memory data reliability in a best-effort way. Further, since losing in-memory data will not affect user persistent data, we propose to provide memory reliability in a best-effort manner.

We use one primary mComponent, one secondary mComponent, and a backup file in sComponent for each vma. A mComponent can serve as the primary for some vma and the secondary for others. The primary stores all memory data and metadata. LegoOS maintains a small append-only log at the secondary mComponent and also replicates the vma tree there. When pComponent flushes a dirty ExCache line, LegoOS sends the data

to both primary and secondary in parallel (step ⓐ and ⓑ in Figure 6) and waits for both to reply (ⓒ and ⓓ). In the background, the secondary mComponent flushes the backup log to a sComponent, which writes it to an append-only file.

If the flushing of a backup log to sComponent is slow and the log is full, we will skip replicating application memory. If the primary fails during this time, LegoOS simply reports an error to application. Otherwise when a primary mComponent fails, we can recover memory content by replaying the backup logs on sComponent and in the secondary mComponent. When a secondary mComponent fails, we do not reconstruct anything and start replicating to a new backup log on another mComponent.

5 LegoOS Implementation

We implemented LegoOS in C on the x86-64 architecture. LegoOS can run on commodity, off-the-shelf machines and support most commonly-used Linux system call APIs. Apart from being a proof-of-concept of the splitkernel OS architecture, our current LegoOS implementation can also be used on existing datacenter servers to reduce the energy cost, with the help of techniques like Zombieland [77]. Currently, LegoOS has 206K SLOC, with 56K SLOC for drivers. LegoOS supports 113 syscalls, 15 pseudo-files, and 10 vectored syscall opcodes. Similar to the findings in [100], we found that implementing these Linux interfaces are sufficient to run many unmodified datacenter applications.

5.1 Hardware Emulation

Since there is no real resource disaggregation hardware, we emulate disaggregated hardware components using commodity servers by limiting their internal hardware usages. For example, to emulate controllers for mComponents and sComponents, we limit the usable cores of a server to two. To emulate pComponents, we limit the amount of usable main memory of a server and configure it as LegoOS software-managed ExCache.

5.2 Network Stack

We implemented three network stacks in LegoOS. The first is a customized RDMA-based RPC framework we implemented based on LITE [101] on top of the Mellanox mlx4 InfiniBand driver we ported from Linux. Our RDMA RPC implementation registers physical memory addresses with RDMA NICs and thus eliminates the need for NICs to cache physical-to-virtual memory address mappings [101]. The resulting smaller NIC SRAM can largely reduce the monetary cost of NICs, further saving the total cost of a LegoOS cluster. All LegoOS internal communications use this RPC framework. For best latency, we use one dedicated polling thread at RPC server side to keep polling incoming requests. Other thread(s) (which we call worker threads) execute the actual RPC functions. For each pair of components, we use one physically consecutive memory region at a component to serve as the receive buffer for RPC requests. The RPC client component uses RDMA write with immediate value to directly write into the memory region and the polling thread polls for the immediate value to get the metadata information about the RPC request (*e.g.*, where the request is written to in the memory region). Immediately after getting an incoming request, the polling thread passes it along to a work queue and continues to poll for the next incoming request. Each worker thread checks if the work queue is not empty and if so, gets an RPC request to process. Once it finishes the RPC function, it sends the return value back to the RPC client with an RDMA write to a memory address at the RPC client. The RPC client allocates this memory address for the return value before sending the RPC request and piggy-backs the memory address with the RPC request.

The second network stack is our own implementation of the socket interface directly on RDMA. The final stack is a traditional socket TCP/IP stack we adapted from lwip [35] on our ported e1000 Ethernet driver. Applications can choose between these two socket implementations and use virtual IPs for their socket communication.

5.3 Processor Monitor

We reserve a contiguous physical memory region during kernel boot time and use fixed ranges of memory in this region as ExCache, tags and metadata for these caches, and kernel physical memory. We organize ExCache into virtually indexed sets with a configurable set associativity. Since x86 (and most other architectures) uses hardware-managed TLB and walks page table directly after TLB misses, we have to use paging and the only chance we can trap to OS is at page fault time. We thus use paged memory to emulate ExCache, with each ExCache line being a 4 KB page. A smaller ExCache line size would improve the performance of fetching lines from mComponents but increase the size of ExCache tag array and the overhead of tag comparison.

An ExCache miss causes a page fault and traps to LegoOS. To minimize the overhead of context switches, we use the application thread that faults on a ExCache miss to perform ExCache replacement. Specifically, this thread will identify the set to insert the missing page using its virtual memory address, evict a page in this set if it is full, and if needed, flush a dirty page to mComponent (via a LegoOS RPC call to the owner mComponent of the vRegion this page is in). To minimize the network round trip needed to complete a ExCache miss, we piggy-back the request of dirty page flush and new page fetching in one RPC call when the mComponent to be flushed to and the mComponent to fetch the missing page are the same.

LegoOS maintains an approximate LRU list for each ExCache set and uses a background thread to sweep all entries in ExCache and adjust LRU lists. LegoOS supports two ExCache replacement policies: FIFO and LRU. For FIFO replacement, we simply maintain a FIFO queue for each ExCache set and insert a corresponding entry to the tail of the FIFO queue when an ExCache page is inserted into the set. Eviction victim is chosen as the head of the FIFO queue. For LRU, we use one background thread to sweep all sets of ExCache to adjust their LRU lists. For both policies, we use a per-set lock and lock the FIFO queue (or the LRU list) when making changes to them.

5.4 Memory Monitor

We use regular machines to emulate mComponents by limiting usable cores to a small number (2 to 5 depending on configuration). We dedicate one core to busy poll network requests and the rest for performing memory functionalities. The LegoOS memory monitor performs all its functionalities as handlers of RPC requests from pComponents. The memory monitor handles most of these functionalities locally and sends another RPC request to a sComponent for storage-related functionalities (*e.g.*, when a buffer cache miss happens). LegoOS stores application data, application memory address mappings, vma trees, and vRegion arrays all in the main memory of the emulating machine.

The memory monitor loads an application executable from sComponents to the mComponent, handles application virtual memory address allocation requests, allocates physical memory at the mComponent, and reads/writes data to the mComponent. Our current implementation of memory monitor is purely in software, and we use hash tables to implement the virtual-to-physical address mappings. While we envision future mComponents to implement memory monitors in hardware and to have specialized hardware parts to store address mappings, our current software implementation can still be useful for users that want to build software-managed mComponents.

5.5 Storage Monitor

Since storage is not the focus of the current version of LegoOS, we chose a simple implementation of building storage monitor on top of the Linux *vfs* layer as a loadable Linux kernel module. LegoOS creates a normal file over vfs as the mount partition for each vNode and issues vfs file operations to perform LegoOS storage I/Os. Doing so is sufficient to evaluate LegoOS, while largely saving our implementation efforts on storage device drivers and layered storage protocols. We leave exploring other options of building LegoOS storage monitor to future work.

5.6 Experience and Discussion

We started our implementation of LegoOS from scratch to have a clean design and implementation that can fit the splitkernel model and to evaluate the efforts needed in building different monitors. However, with the vast amount and the complexity of drivers, we decided to port Linux drivers instead of writing our own. We then spent our engineering efforts on an "as needed" base and took shortcuts by porting some of the Linux code. For example, we re-used common algorithms and data structures in Linux to easily port Linux drivers. We believe that being able to support largely unmodified Linux drivers will assist the adoption of LegoOS.

When we started building LegoOS, we had a clear goal of sticking to the principle of "clean separation of functionalities". However, we later found several places where performance could be improved if this principle is relaxed. For example, for the optimization in §4.4.2 to work correctly, pComponent needs to store the address range and permission for anonymous virtual memory regions — memory-related information that otherwise only mComponents need to know. Another example is the implementation of `mremap`. With LegoOS' principle of mComponents handling all memory address allocations, memory monitors will allocate new virtual memory address ranges for `mremap` requests. However, when data in the `mremap` region is in ExCache, LegoOS needs to move it to another set if the new virtual address does not fall into the current set. If mComponents are ExCache-aware, they can choose the new virtual memory address to fall into the same set as the current one. Our strategy is to relax the clean-separation principle only by giving "hints", and only for frequently-accessed, performance-critical operations.

6 Evaluation

This section presents the performance evaluation of LegoOS using micro- and macro-benchmarks and two unmodified real applications. We also quantitatively analyze the failure rate of LegoOS. We ran all experiments on a cluster of 10 machines, each with two Intel Xeon CPU E5-2620 2.40GHz processors, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter; a Mellanox 40 Gbps InfiniBand switch connects all of the machines. The Linux version we used for comparison is v4.9.47.

6.1 Micro- and Macro-benchmark Results

Network performance. Network communication is at the core of LegoOS' performance. Thus, we evaluate LegoOS' network performance first before evaluating LegoOS as a whole. Figure 7 plots the average latency of sending messages with socket-over-InfiniBand (Linux-

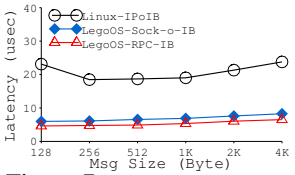


Figure 7: **Network Latency.**

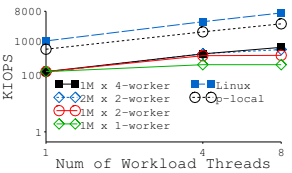


Figure 8: **Memory Throughput.**

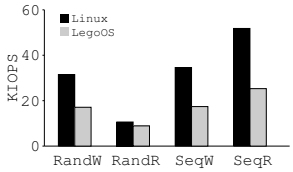


Figure 9: **Storage Throughput.**

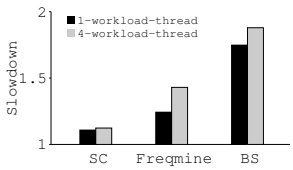


Figure 10: **PARSEC Results.** *SC: StreamClsuter. BS: BlackScholes.*

IPoIB) in Linux, LegoOS' implementation of socket on top of InfiniBand (LegoOS-Sock-o-IB), and LegoOS' implementation of RPC over InfiniBand (LegoOS-RPC-IB). LegoOS uses LegoOS-RPC-IB for all its internal network communication across components and uses LegoOS-Sock-o-IB for all application-initiated socket network requests. Both LegoOS' networking stacks significantly outperform Linux's.

Memory performance. Next, we measure the performance of mComponent using a multi-threaded user-level micro-benchmark. In this micro-benchmark, each thread performs one million sequential 4 KB memory loads in a heap. We use a huge, empty ExCache (32 GB) to run this test, so that each memory access can generate an ExCache (cold) miss and go to the mComponent.

Figure 8 compares LegoOS' mComponent performance with Linux's single-node memory performance using this workload. We vary the number of per-mComponent worker threads from 1 to 8 with one and two mComponents (only showing representative configurations in Figure 8). In general, using more worker threads per mComponent and using more mComponents both improve throughput when an application has high parallelism, but the improvement largely diminishes after the total number of worker threads reaches four. We also evaluated the optimization technique in § 4.4.2 (*p-local* in Figure 8). As expected, bypassing mComponent accesses with p-local lines significantly improves memory access performance. The difference between p-local and Linux demonstrates the overhead of trapping to LegoOS kernel and setting up ExCache.

Storage performance. To measure the performance of LegoOS' storage system, we ran a single-thread micro-benchmark that performs sequential and random 4 KB read/write to a 25 GB file on a Samsung PM1725s NVMe SSD (the total amount of data accessed is 1 GB). For write workloads, we issue an *fsync* after each *write* to test the performance of writing all the way to the SSD.

Figure 9 presents the throughput of this workload on LegoOS and on single-node Linux. For LegoOS, we use one mComponent to store the buffer cache of this file and initialize the buffer cache to empty so that file I/Os can go to the sComponent (Linux also uses an empty buffer cache). Our results show that Linux's performance is determined by the SSD's read/write bandwidth. LegoOS' random read performance is close to Linux, since network cost is relatively low compared to the SSD's random read performance. With better SSD sequential read performance, network cost has a higher impact. LegoOS' write-and-fsync performance is worse than Linux because LegoOS requires one RTT between pComponent and mComponent to perform write and two RTTs (pComponent to mComponent, mComponent to sComponent) for fsync.

PARSEC results. We evaluated LegoOS with a set of workloads from the PARSEC benchmark suite [22], including BlackScholes, Freqmine, and StreamCluster. These workloads are a good representative of compute-intensive datacenter applications, ranging from machine-learning algorithms to streaming processing ones. Figure 10 presents the slowdown of LegoOS over single-node Linux with enough memory for the entire application working sets. LegoOS uses one pComponent with 128 MB ExCache, one mComponent with one worker thread, and one sComponent for all the PARSEC tests. For each workload, we tested one and four workload threads. StreamCluster, a streaming workload, performs the best because of its batching memory access pattern (each batch is around 110 MB). BlackScholes and Freqmine perform worse because of their larger working sets (630 MB to 785 MB). LegoOS performs worse with higher workload threads, because the single worker thread at the mComponent becomes the bottleneck to achieving higher throughput.

6.2 Application Performance

We evaluated LegoOS' performance with two real, unmodified applications, TensorFlow [4] and Phoenix [85], a single-node multi-threaded implementation of MapReduce [32]. TensorFlow's experiments use the Cifar-10 dataset [2] and Phoenix's use a Wikipedia dataset [3]. Unless otherwise stated, the base configuration used for all TensorFlow experiments is 256 MB 64-way ExCache, one pComponent, one mComponent, and one sComponent. The base configuration for Phoenix is the same as TensorFlow's with the exception that the base ExCache size is 512 MB. The total amount of virtual memory addresses touched in TensorFlow is 4.4 GB (1.75 GB for Phoenix). The total working sets of the TensorFlow and Phoenix execution are 0.9 GB and 1.7 GB. Our default

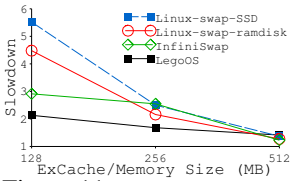


Figure 11: **TensorFlow Perf.**

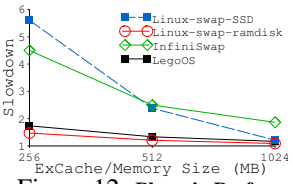


Figure 12: **Phoenix Perf.**



Figure 13: **ExCache Mgmt.**

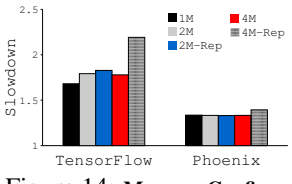


Figure 14: **Memory Config.**

ExCache sizes are set as roughly 25% of total working sets. We ran both applications with four threads.

Impact of ExCache size on application performance. Figures 11 and 12 plot the TensorFlow and Phoenix run time comparison across LegoOS, a remote swapping system (InfiniSwap [47]), a Linux server with a swap file in a local high-end NVMe SSD, and a Linux server with a swap file in local ramdisk. All values are calculated as a slowdown to running the applications on a Linux server that have enough local resources (main memory, CPU cores, and SSD). For systems other than LegoOS, we change the main memory size to the same size of ExCache in LegoOS, with rest of the memory on swap file. With around 25% working set, LegoOS only has a slowdown of 1.68× and 1.34× for TensorFlow and Phoenix compared to a monolithic Linux server that can fit all working sets in its main memory.

LegoOS' performance is significantly better than swapping to SSD and to remote memory largely because of our efficiently-implemented network stack, simplified code path compared with Linux paging subsystem, and the optimization technique proposed in §4.4.2. Surprisingly, it is similar or even better than swapping to local memory, even when LegoOS' memory accesses are across network. This is mainly because ramdisk goes through buffer cache and incurs memory copies between the buffer cache and the in-memory swap file.

LegoOS' performance results are not easy to achieve and we went through rounds of design and implementation refinement. Our network stack and RPC optimizations yield a total improvement of up to 50%. For example, we made all RPC server (mComponent's) replies *unsignaled* to save mComponent' processing time and to increase its request handling throughput. Another optimization we did is to piggy-back dirty cache line flush and cache miss fill into one RPC. The optimization of the first anonymous memory access (§4.4.2) improves LegoOS' performance further by up to 5%.

ExCache Management. Apart from its size, how an ExCache is managed can also largely affect application performance. We first evaluated factors that could affect ExCache hit rate and found that higher associativity improves hit rate but the effect diminishes when going beyond 512-way. We then focused on evaluating the miss cost of ExCache, since the miss path is handled by LegoOS in our design. We compare the two eviction policies LegoOS supports (FIFO and LRU), two implementations of finding an empty line in an ExCache set (linearly scan a free bitmap and fetching the head of a free list), and one network optimization (piggyback flushing a dirty line with fetching the missing line).

Figure 13 presents these comparisons with one and four mComponent worker threads. All tests run the Cifar-10 workload on TensorFlow with 256 MB 64-way ExCache, one mComponent, and one sComponent. Using bitmaps for this ExCache configuration is always worse than using free lists because of the cost to linearly scan a whole bitmap, and bitmaps perform even worse with higher associativity. Surprisingly, FIFO performs better than LRU in our tests, even when LRU utilizes access locality pattern. We attributed LRU's worse performance to the lock contention it incurs; the kernel background thread sweeping the ExCache locks an LRU list when adjusting the position of an entry in it, while ExCache miss handler thread also needs to lock the LRU list to grab its head. Finally, the piggyback optimization works well and the combination of FIFO, free list, and piggyback yields the best performance.

Number of mComponents and replication. Finally, we study the effect of the number of mComponents and memory replication. Figure 14 plots the performance slowdown as the number of mComponents increases from one to four. Surprisingly, using more mComponents lowers application performance by up to 6%. This performance drop is due to the effect of ExCache piggyback optimization. When there is only one mComponent, flushes and misses are all between the pComponent and this mComponent, thus enabling piggyback on every flush. However, when there are multiple mComponents, LegoOS can only perform piggyback when flushes and misses are to the same mComponent.

We also evaluated LegoOS' memory replication performance in Figure 14. Replication has a performance overhead of 2% to 23% (there is a constant 1 MB space overhead to store the backup log). LegoOS uses the same application thread to send the replica data to the backup mComponent and then to the primary mComponent, resulting in the performance lost.

Running multiple applications together. All our experiments so far run only one application at a time. Now we evaluate how multiple applications perform when running them together on a LegoOS cluster. We use a simple scenario of running one TensorFlow instance and one Phoenix instance together in two settings: 1) two pCom-

	Processor	Disk	Memory	NIC	Power	Other	Monolithic	LegoOS
MTTF (year)	204.3	33.1	289.9	538.8	100.5	27.4	5.8	6.8 - 8.7

Table 1: **Mean Time To Failure Analysis.** *MTTF numbers of devices (columns 2 to 7) are obtained from [90] and MTTF values of monolithic server and LegoOS are calculated using the per-device MTTF numbers.*

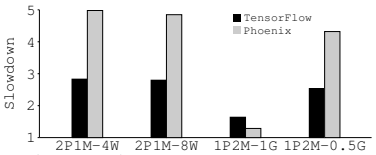


Figure 15: **Multiple Applications.**

ponents each running one instance, both accessing one mComponent(2P1M), and 2) one pComponent running two instances and accessing two mComponents (1P2M). Both settings use one sComponent. Figure 15 presents the runtime slowdown results. We also vary the number of mComponent worker threads for the 2P1M setting and the amount of ExCache for the 1P2M setting. With 2P1M, both applications suffer from a performance drop because their memory access requests saturate the single mComponent. Using more worker threads at the mComponent improves the performance slightly. For 1P2M, application performance largely depends on ExCache size, similar to our findings with single-application experiments.

6.3 Failure Analysis

Finally, we provide a qualitative analysis on the failure rate of a LegoOS cluster compared to a monolithic server cluster. Table 1 summarizes our analysis. To measure the failure rate of a cluster, we use the metric Mean Time To (hardware) Failure (MTTF), the mean time to the failure of a server in a monolithic cluster or a component in a LegoOS cluster. Since the only real per-device failure statistics we can find are the mean time to hardware replacement in a cluster [90], the MTTF we refer to in this study indicates the mean time to the type of hardware failures that require replacement. Unlike traditional MTTF analysis, we are not able to include transient failures.

To calculate MTTF of a monolithic server, we first obtain the replacement frequency of different hardware devices in a server (CPU, memory, disk, NIC, motherboard, case, power supply, fan, CPU heat sink, and other cables and connections) from the real world (the COM1 and COM2 clusters in [90]). For LegoOS, we envision every component to have a NIC and a power supply, and in addition, a pComponent to have CPU, fan, and heat sink, an mComponent to have memory, and an sComponent to have a disk. We further assume both a monolithic server and a LegoOS component to fail when any hardware devices in them fails and the devices in them fail independently. Thus, the MTTF can be calculated using the harmonic mean (*HM*) of the MTTF of each device.

$$MTTF = \frac{HM_{i=0}^{n}(MTTF_i)}{n} \quad (1)$$

where n includes all devices in a machine/component.

Further, when calculating MTTF of LegoOS, we estimate the amount of components needed in LegoOS to run the same applications as a monolithic cluster. Our estimated worst case for LegoOS is to use the same amount of hardware devices (*i.e.*, assuming same resource utilization as monolithic cluster). LegoOS' best case is to achieve full resource utilization and thus requiring only about half of CPU and memory resources (since average CPU and memory resource utilization in monolithic server clusters is around 50% [10, 45]).

With better resource utilization and simplified hardware components (*e.g.*, no motherboard), LegoOS improves MTTF by 17% to 49% compared to an equivalent monolithic server cluster.

7 Related Work

Hardware Resource Disaggregation. There have been a few hardware disaggregation proposals from academia and industry, including Firebox [15], HP "The Machine" [40, 48], dRedBox [56], and IBM Composable System [28]. Among them, dRedBox and IBM Composable System package hardware resources in one big case and connect them with buses like PCIe. The Machine's scale is a rack and it connects SoCs with NVMs with a specialized coherent network. FireBox is an early-stage project and is likely to use high-radix switches to connect customized devices. The disaggregated cluster we envision to run LegoOS on is one that hosts hundreds to thousands of non-coherent, heterogeneous hardware devices, connected with a commodity network.

Memory Disaggregation and Remote memory. Lim *et al.* first proposed the concept of hardware disaggregated memory with two models of disaggregated memory: using it as network swap device and transparently accessing it through memory instructions [63, 64]. Their hardware models still use a monolithic server at the local side. LegoOS' hardware model separates processor and memory completely.

Another set of recent projects utilize remote memory without changing monolithic servers [6, 34, 47, 74, 79, 93]. For example, InfiniSwap [47] transparently swaps local memory to remote memory via RDMA. These remote memory systems help improve the memory resource packing in a cluster. However, as discussed in §2, unlike LegoOS, these solutions cannot solve other lim-

itations of monolithic servers like the lack of hardware heterogeneity and elasticity.

Storage Disaggregation. Cloud vendors usually provision storage at different physical machines [13, 103, 108]. Remote access to hard disks is a common practice, because their high latency and low throughput can easily hide network overhead [61, 62, 70, 105]. While disaggregating high-performance flash is a more challenging task [38, 59]. Systems such as ReFlex [60], Decibel [73], and PolarFS [25], tightly integrate network and storage layers to minimize software overhead in the face of fast hardware. Although storage disaggregation is not our main focus now, we believe those techniques can be realized in future LegoOS easily.

Multi-Kernel and Multi-Instance OSes. Multi-kernel OSes like Barrelfish [21, 107], Helios [76], Hive [26], and fos [106] run a small kernel on each core or programmable device in a monolithic server, and they use message passing to communicate across their internal kernels. Similarly, multi-instance OSes like Popcorn [17] and Pisces [83] run multiple Linux kernel instances on different cores in a machine. Different from these OSes, LegoOS runs on and manages a distributed set of hardware devices; it manages distributed hardware resources using a two-level approach and handles device failures (currently only mComponent). In addition, LegoOS differs from these OSes in how it splits OS functionalities, where it executes the split kernels, and how it performs message passing across components. Different from multi-kernels' message passing mechanisms which are performed over buses or using shared memory in a server, LegoOS' message passing is performed using a customized RDMA-based RPC stack over InfiniBand or RoCE network. Like LegoOS, fos [106] separates OS functionalities and run them on different processor cores that share main memory. Helios [76] runs *satellite kernels* on heterogeneous cores and programmable NICs that are not cache-coherent. We took a step further by disseminating OS functionalities to run on individual, network-attached hardware devices. Moreover, LegoOS is the first OS that separates memory and process management and runs virtual memory system completely at network-attached memory devices.

Distributed OSes. There have been several distributed OSes built in late 80s and early 90s [14, 16, 19, 27, 82, 86, 96, 97]. Many of them aim to appear as a single machine to users and focus on improving inter-node IPCs. Among them, the most closely related one is Amoeba [96, 97]. It organizes a cluster into a shared process pool and disaggregated specialized servers. Unlike Amoeba, LegoOS further separates memory from processors and different hardware components are loosely coupled and can be heterogeneous instead of as a homogeneous pool. There are also few emerging proposals to build distributed OSes in datacenters [54, 91], *e.g.*, to reduce the performance overhead of middleware. LegoOS achieves the same benefits of minimal middleware layers by only having LegoOS as the system management software for a disaggregated cluster and using the lightweight vNode mechanism.

8 Discussion and Conclusion

We presented LegoOS, the first OS designed for hardware resource disaggregation. LegoOS demonstrated the feasibility of resource disaggregation and its advantages in better resource packing, failure isolation, and elasticity, all without changing Linux ABIs. LegoOS and resource disaggregation in general can help the adoption of new hardware and thus encourage more hardware and system software innovations.

LegoOS is a research prototype and has a lot of room for improvement. For example, we found that the amount of parallel threads an mComponent can use to process memory requests largely affect application throughput. Thus, future developers of real mComponents can consider use large amount of cheap cores or FPGA to implement memory monitors in hardware.

We also performed an initial investigation in load balancing and found that memory allocation policies across mComponents can largely affect application performance. However, since we do not support memory data migration yet, the benefit of our load-balancing mechanism is small. We leave memory migration for future work. In general, large-scale resource management of a disaggregated cluster is an interesting and important topic, but is outside of the scope of this paper.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Michael Swift for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We are also thankful to Sumukh H. Ravindra for his contribution in the early stage of this work.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 1719215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Open Compute Project (OCP). http://www.opencompute.org/.

[2] The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[3] Wikipedia dump. https://dumps.wikimedia.org/.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[5] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar. A Many-core Architecture for In-memory Data Processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC '18)*.

[7] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[9] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[10] Alibaba. Alibaba Production Cluster Trace Data. https://github.com/alibaba/clusterdata.

[11] Amazon. Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[12] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[13] Amazon. Amazon EC2 Root Device Volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[14] Y. Artsy, H.-Y. Chang, and R. Finkel. Interprocess communication in charlotte. *IEEE Software*, Jan 1987.

[15] K. Asanovi. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[16] A. Barak and R. Wheeler. *MOSIX: An integrated unix for multiprocessor workstations*. International Computer Science Institute, 1988.

[17] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*.

[18] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12), December 2007.

[19] F. Baskett, J. H. Howard, and J. T. Montague. Task Communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP '77)*.

[20] A. Basu, M. D. Hill, and M. M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[21] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*.

[23] M. N. Bojnordi and E. Ipek. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[24] Cache Coherent Interconnect for Accelerators, 2018. https://www.ccixconsortium.com/.

[25] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment (VLDB '18)*.

[26] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*.

[27] D. Cheriton. The V Distributed System. *Commun. ACM*, 31(3), March 1988.

[28] I.-H. Chung, B. Abali, and P. Crumley. Towards a Composable Computer System. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia '18)*.

[29] Cisco, EMC, and Intel. The Performance Impact of NVMe and NVMe over Fabrics. http://www.snia.org/sites/default/files/NVMe_Webcast_Slides_Final.1.pdf.

[30] P. Costa. Towards rack-scale computing: Challenges and opportunities. In *First International Workshop on Rack-scale Computing (WRSC '14)*.

[31] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. In *Proceedings of the ACM SIGCOMM 2015 Conference on SIGCOMM (SIGCOMM '15)*.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI '04)*.

[33] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[34] Dragojević, Aleksandar and Narayanan, Dushyanth and Hodson, Orion and Castro, Miguel. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*.

[35] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2001.

[36] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*.

[37] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Princi-*

ples (SOSP '95).

[38] Facebook. Introducing Lightning: A flexible NVMe JBOF. https://code.fb.com/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof/.

[39] Facebook. Wedge 100: More open and versatile than ever. https://code.fb.com/networking-traffic/wedge-100-more-open-and-versatile-than-ever/.

[40] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*.

[41] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[42] GenZ Consortium. http://genzconsortium.org/.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*.

[44] J. R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)*.

[45] Google. Google Production Cluster Trace Data. https://github.com/google/cluster-data.

[46] Google. GPUs are now available for Google Compute Engine and Cloud Machine Learning. https://cloudplatform.googleblog.com/2017/02/GPUs-are-now-available-for-Google-Compute-Engine-and-Cloud-Machine-Learning.html.

[47] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[48] Hewlett-Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/.

[49] Intel. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint.

[50] Intel. Intel Omni-Path Architecture. https://tinyurl.com/ya3x4ktd.

[51] Intel. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[52] Intel, 2018. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/.

[53] JEDEC. HIGH BANDWIDTH MEMORY (HBM) DRAM JESD235A. https://www.jedec.org/standards-documents/docs/jesd235a.

[54] W. John, J. Halen, X. Cai, C. Fu, T. Holmberg, V. Katardjiev, M. Sedaghat, P. Sköldström, D. Turull, V. Yadhav, and J. Kempf. Making Cloud Easy: Design Considerations and First Components of a Distributed Operating System for Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. D. Mike Daley, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, H. K. Alexander Kaplan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, M. R. Jonathan Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, A. S. Dan Steinberg, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[56] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE '16)*.

[57] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[58] S. Kaxiras and A. Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[59] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[60] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[61] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*.

[62] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*.

[63] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[64] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*.

[65] D. Meisner, B. T. Gold, and T. F. Wenisch. The powernap server architecture. *ACM Trans. Comput. Syst.*, February 2011.

[66] Mellanox. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card.

[67] Mellanox. Mellanox BuleField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic.

[68] Mellanox. Mellanox Innova Adapters. http://www.mellanox.com/page/programmable_network_adapters?mtag=&mtag=programmable_adapter_cards.

[69] Mellanox. Rdma aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[70] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring,

B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*.

[71] D. Minturn. NVM Express Over Fabrics. 11th Annual OpenFabrics International OFS Developers' Workshop, March 2015.

[72] T. P. Morgan. Enterprises Get On The Xeon Phi Roadmap. https://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/.

[73] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[74] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*.

[75] Netronome. Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[76] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[77] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[78] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[79] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*.

[80] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*.

[81] Open Coherent Accelerator Processor Interface, 2018. https://opencapi.org/.

[82] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *Computer*, 21(2), February 1988.

[83] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*.

[84] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*.

[85] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA '07)*.

[86] R. F. Rashid and G. G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*.

[87] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[88] S. M. Rumble. Infiniband Verbs Performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance.

[89] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, 1985.

[90] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*.

[91] M. Schwarzkopf, M. P. Grosvenor, and S. Hand. New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*.

[92] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

[93] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[94] M. Silberstein. Accelerators in data centers: the systems perspective. https://www.sigarch.org/accelerators-in-data-centers-the-systems-perspective/.

[95] A. J. Smith. Cache Memories. *ACM Comput. Surv.*, 14(3), September 1982.

[96] A. S. Tanenbaum, M. F. Kaashoek, R. Van Renesse, and H. E. Bal. The Amoeba Distributed Operating System&Mdash;a Status Report. *Comput. Commun.*, 14(6), August 1991.

[97] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12), December 1990.

[98] E. Technologies. NVMe Storage Accelerator Series. https://www.everspin.com/nvme-storage-accelerator-series.

[99] S. Thomas, G. M. Voelker, and G. Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[100] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[101] S.-Y. Tsai and Y. Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[102] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys '15)*.

[103] VMware. Virtual SAN. https://www.vmware.com/products/vsan.html.

[104] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*.

[105] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HotOS '05)*.

[106] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and

A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*.

[107] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*.

[108] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

The benefits and costs of writing a POSIX kernel in a high-level language

Cody Cutler, M. Frans Kaashoek, Robert T. Morris
MIT CSAIL

Abstract

This paper presents an evaluation of the use of a high-level language (HLL) with garbage collection to implement a monolithic POSIX-style kernel. The goal is to explore if it is reasonable to use an HLL instead of C for such kernels, by examining performance costs, implementation challenges, and programmability and safety benefits.

The paper contributes Biscuit, a kernel written in Go that implements enough of POSIX (virtual memory, `mmap`, TCP/IP sockets, a logging file system, `poll`, etc.) to execute significant applications. Biscuit makes liberal use of Go's HLL features (closures, channels, maps, interfaces, garbage collected heap allocation), which subjectively made programming easier. The most challenging puzzle was handling the possibility of running out of kernel heap memory; Biscuit benefited from the analyzability of Go source to address this challenge.

On a set of kernel-intensive benchmarks (including NGINX and Redis) the fraction of kernel CPU time Biscuit spends on HLL features (primarily garbage collection and thread stack expansion checks) ranges up to 13%. The longest single GC-related pause suffered by NGINX was 115 microseconds; the longest observed sum of GC delays to a complete NGINX client request was 600 microseconds. In experiments comparing nearly identical system call, page fault, and context switch code paths written in Go and C, the Go version was 5% to 15% slower.

1 Introduction

The default language for operating system kernels is C: Linux, macOS, and Windows all use C. C is popular for kernels because it can deliver high performance via flexible low-level access to memory and control over memory management (allocation and freeing). C, however, requires care and experience to use safely, and even then low-level bugs are common. For example, in 2017 at least 50 Linux kernel security vulnerabilities were reported that involved buffer overflow or use-after-free bugs in C code [34].

High-level languages (HLLs) provide type- and memory-safety and convenient abstractions such as threads. Many HLLs provide garbage collection to further reduce programmer burden and memory bugs. It is well-known that HLLs can be used in kernels: multiple kernels have been written in HLLs, often as platforms to explore innovative ideas (§2). On the other hand, leading OS designers have been skeptical that HLLs' memory management and abstractions are compatible with high-performance production kernels [51][47, p. 71].

While it would probably not make sense to re-write an existing C kernel in an HLL, it is worth considering what languages new kernel projects should use. Since kernels impose different constraints and requirements than typical applications, it makes sense to explore this question in the context of a kernel.

We built a new kernel, Biscuit, written in Go [15] for x86-64 hardware. Go is a type-safe language with garbage collection. Biscuit runs significant existing applications such as NGINX and Redis without source modification by exposing a POSIX-subset system call interface. Supported features include multi-core, kernel-supported user threads, futexes, IPC, mmap, copy-on-write fork, vnode and name caches, a logging file system, and TCP/IP sockets. Biscuit implements two significant device drivers in Go: one for AHCI SATA disk controllers and one for Intel 82599-based Ethernet controllers. Biscuit has nearly 28 thousand lines of Go, 1546 lines of assembler, and no C. We report lessons learned about use of Go in Biscuit, including ways in which the language helped development, and situations in which it was less helpful.

In most ways the design of Biscuit is that of a traditional monolithic POSIX/Unix kernel, and Go was a comfortable language for that approach. In one respect the design of Biscuit is novel: its mechanism for coping with kernel heap exhaustion. We use static analysis of the Biscuit source to determine how much heap memory each system call (and other kernel activity) might need, and each system call waits (if needed) when it starts until it can reserve that much heap. Once a system call is allowed to continue, its allocations are guaranteed to succeed without blocking. This obviates the need for complex allocation failure recovery or deadlock-prone waiting for free memory in the allocator. The use of an HLL that is conducive to static analysis made this approach possible.

We run several kernel-intensive applications on Biscuit and measure the effects of Go's type safety and garbage collection on kernel performance. For our benchmarks, GC costs up to 3% of CPU. For NGINX, the longest single

GC-related pause was 115 microseconds, and the longest a single NGINX client request was delayed (by many individual pauses) was a total of 600 microseconds. Other identifiable HLL performance costs amount to about 10% of CPU.

To shed light on the specific question of C versus Go performance in the kernel, we modify Biscuit and a C kernel to have nearly identical source-level code paths for two benchmarks that stress system calls, page faults, and context switches. The C versions are about 5% and 15% faster than the Go versions.

Finally, we compare the performance of Biscuit and Linux on our kernel-intensive application benchmarks, finding that Linux is up to 10% faster than Biscuit. This result is not very illuminating about choice of language, since performance is also affected by differences in the features, design and implementation of Biscuit and Linux. However, the results do provide an idea of whether the absolute performance of Biscuit is in the same league as that of a C kernel.

In summary, the main contributions of this paper are: (1) Biscuit, a kernel written in Go with good performance; (2) a novel scheme for coping with kernel heap exhaustion; (3) a discussion of qualitative ways in which use of an HLL in a kernel was and was not helpful; (4) measurements of the performance tax imposed by use of an HLL; and (5) a direct Go-vs-C performance comparison of equivalent code typical of that found in a kernel.

This paper does not draw any top-level conclusion about C versus an HLL as a kernel implementation language. Instead, it presents experience and measurements that may be helpful for others making this decision, who have specific goals and requirements with respect to programmability, safety and performance. Section 9 summarizes the key factors in this decision.

2 Related work

Biscuit builds on multiple areas of previous work: high-level languages in operating systems, high-level systems programming languages, and memory allocation in the kernel. As far as we know the question of the impact of language choice on kernel performance, all else being equal, has not been explored.

Kernels in high-level languages. The Pilot [44] kernel and the Lisp machine [17] are early examples of use of a high-level language (Mesa [14] and Lisp, respectively) in an operating system. Mesa lacked garbage-collection, but it was a high-priority requirement for its successor language Cedar [48]. The Lisp machine had a real-time garbage collector [5].

A number of research kernels are written in high-level languages (e.g., Taos [49], Spin [7], Singularity [23], J-kernel [19], and KaffeOS [3, 4], House [18], the Mirage unikernel [29], and Tock [27]). The main thrust of these projects was to explore new ideas in operating system architecture, often enabled by the use of a type-safe high-level language. While performance was often a concern, usually the performance in question related to the new ideas, rather than to the choice of language. Singularity quantified the cost of hardware and software isolation [22], which is related to the use of a HLL, but didn't quantify the cost of safety features of a HLL language, as we do in §8.4.

High-level systems programming languages. A number of systems-oriented high-level programming languages with type safety and garbage collection seem suitable for kernels, including Go, Java, C#, and Cyclone [25] (and, less recently, Cedar [48] and Modula-3 [37]). Other systems HLLs are less compatible with existing kernel designs. For example, Erlang [2] is a "shared-nothing" language with immutable objects, which would likely result in a kernel design that is quite different from traditional C shared-memory kernels.

Frampton et al. introduce a framework for language extensions to support low-level programming features in Java, applying it to a GC toolkit [13]. Biscuit's goal is efficiency for kernels without modifying Go. Kernels have additional challenges such as dealing with user/kernel space, page tables, interrupts, and system calls.

A number of new languages have recently emerged for systems programming: D [11], Nim(rod) [42], Go [15], and Rust [36]. There are a number of kernels in Rust [12, 26, 27, 28, 39, 50], but none were written with the goal of comparing with C as an implementation language. Gopher OS is a Go kernel with a similar goal as Biscuit, but the project is at an early stage of development [1]. Other Go kernels exists but they don't target the questions that Biscuit answers. For example, Clive [6] is a unikernel and doesn't run on the bare metal. The Ethos OS uses C for the kernel and Go for user-space programs, with a design focused on security [41]. gVisor is a user-space kernel, written in Go, that implements a substantial portion of the Linux system API to sandbox containers [16].

Memory allocation. There is no consensus about whether a systems programming language should have automatic garbage-collection. For example, Rust is partially motivated by the idea that garbage collection cannot be made efficient; instead, the Rust compiler analyzes the program to partially automate freeing of memory. This approach can make sharing data among multiple threads or closures awkward [26].

Concurrent garbage collectors [5, 24, 30] reduce pause times by collecting while the application runs. Go 1.10 has such a collector [21], which Biscuit uses.

Several papers have studied manual memory allocation versus automatic garbage collection [20, 52], focusing on heap headroom memory's effect in reducing garbage collection costs in user-level programs. Headroom is also important for Biscuit's performance (§5 and §8.6).

Rafkind et al. added garbage collection to parts of Linux through automatic translation of C source [43]. The authors observe that the kernel environment made this task difficult and adapted a fraction of a uniprocessor Linux kernel to be compatible with garbage collection. Biscuit required a fresh start in a new language, but as a result required less programmer effort for GC compatibility and benefited from a concurrent and parallel collector.

Linux's slab allocators [8] are specifically tuned for use in the kernel; they segregate free objects by type to avoid re-initialization costs and fragmentation. A hypothesis in the design of Biscuit is that Go's single general-purpose allocator and garbage collector are suitable for a wide range of different kernel objects.

Kernel heap exhaustion. All kernels have to cope with the possibility of running out of memory for the kernel heap. Linux optimistically lets system calls proceed up until the point where an allocation fails. In some cases code waits and re-tries the allocation a few times, to give an "out-of-memory" killer thread time to find and destroy an abusive process to free memory. However, the allocating thread cannot generally wait indefinitely: it may hold locks, so there is a risk of deadlock if the victim of the killer thread is itself waiting for a lock [9]. As a result Linux system calls must contain code to recover from allocation failures, undoing any changes made so far, perhaps unwinding through many function calls. This undo code has a history of bugs [10]. Worse, the final result will be an error return from a system call. Once the heap is exhausted, any system call that allocates will likely fail; few programs continue to operate correctly in the face of unexpected errors from system calls, so the end effect may be application-level failure even if the kernel code handles heap exhaustion correctly.

Biscuit's reservation approach yields simpler code than Linux's. Biscuit kernel heap allocations do not fail (much as with Linux's contentious "too small to fail" rule [9, 10]), eliminating a whole class of complex error recovery code. Instead, each Biscuit system call reserves kernel heap memory when it starts (waiting if necessary), using a static analysis system to decide how much to reserve. Further, Biscuit applications don't see system call failures when the heap is exhausted; instead, they see delays.

3 Motivation

This section outlines our view of the main considerations in the choice between C and an HLL for the kernel.

3.1 Why C?

A major reason for C's popularity in kernels is that it supports low-level techniques that can help performance, particularly pointer arithmetic, easy escape from type enforcement, explicit memory allocation, and custom allocators [51][47, p. 71]. There are other reasons too (e.g. C can manipulate hardware registers and doesn't depend on a complex runtime), but performance seems most important.

3.2 Why an HLL?

The potential benefits of high-level languages are well understood. Automatic memory management reduces programmer effort and use-after-free bugs; type-safety detects bugs; runtime typing and method dispatch help with abstraction; and language support for threads and synchronization eases concurrent programming.

Certain kinds of bugs seem much less likely in an HLL than in C: buffer overruns, use-after-free bugs [40], and bugs caused by reliance on C's relaxed type enforcement. Even C code written with care by expert programmers has C-related bugs [40]. The CVE database for the Linux kernel [34] lists 40 execute-code vulnerabilities for 2017 which would be wholly or partially ameliorated by use of an HLL (see §8.2).

Use-after-free bugs are notoriously difficult to debug, yet occur often enough that the Linux kernel includes a memory checker that detects some use-after-free and buffer overrun bugs at runtime [46]. Nevertheless, Linux developers routinely discover and fix use-after-free bugs: Linux has at least 36 commits from January to April of 2018 for the specific purpose of fixing use-after-free bugs.

Another area of kernel programming that would benefit from HLLs is concurrency. Transient worker threads can be cumbersome in C because the code must decide when the last thread has stopped using any shared objects that need to be freed; this is easier in a garbage collected language.

However, use of a garbage-collected HLL is not free. The garbage collector and safety checks consume CPU time and can cause delays; the expense of high-level features may deter their use; the language's runtime layer hides important mechanisms such as memory allocation; and enforced abstraction and safety may reduce developers' implementation options.

4 Overview

Biscuit's main purpose is to help evaluate the practicality of writing a kernel in a high-level language. Its design is similar to common practice in monolithic UNIX-like kernels, to facilitate comparison. Biscuit runs on 64-bit x86 hardware and is written in Go. It uses a modified version of the Go 1.10 runtime implementation; the runtime is

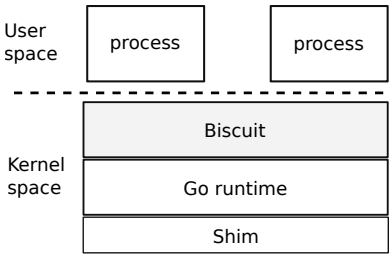


Figure 1: Biscuit's overall structure.

written in Go with some assembly. Biscuit adds more assembly to handle boot and entry/exit for system calls and interrupts. There is no C. This section briefly describes Biscuit's components, focusing on areas in which use of Go affected the design and implementation.

Boot and Go Runtime. The boot block loads Biscuit, the Go runtime, and a "shim" layer (as shown in Figure 1). The Go runtime, which we use mostly unmodified, expects to be able to call an underlying kernel for certain services, particularly memory allocation and control of execution contexts (cores, or in Go terminology, threads). The shim layer provides these functions, since there is no underlying kernel. Most of the shim layer's activity occurs during initialization, for example to pre-allocate memory for the Go kernel heap.

Processes and Kernel Goroutines. Biscuit provides user processes with a POSIX interface: `fork`, `exec`, and so on, including kernel-supported threads and futexes. A user process has one address space and one or more threads. Biscuit uses hardware page protection to isolate user processes. A user program can be written in any language; we have implemented them only in C and C++ (not Go). Biscuit maintains a kernel goroutine corresponding to each user thread; that goroutine executes system calls and handlers for page faults and exceptions for the user thread. "goroutine" is Go's name for a thread, and in this paper refers only to threads running inside the kernel.

Biscuit's runtime schedules the kernel goroutines of user processes, each executing its own user thread in user-mode when necessary. Biscuit uses timer interrupts to switch pre-emptively away from user threads. It relies on pre-emption checks generated by the Go compiler to switch among kernel goroutines.

Interrupts. A Biscuit device interrupt handler marks an associated device-driver goroutine as runnable and then returns, as previous kernels have done [35, 45]. Interrupt handlers cannot do much more without risk of deadlock, because the Go runtime does not turn off interrupts during sensitive operations such as goroutine context switch.

Handlers for system calls and faults from user space can execute any Go code. Biscuit executes this code in the context of the goroutine that is associated with the current user thread.

Multi-Core and Synchronization. Biscuit runs in parallel on multi-core hardware. It guards its data structures using Go's mutexes, and synchronizes using Go's channels and condition variables. The locking is fine-grained enough that system calls from threads on different cores can execute in parallel in many common situations, for example when operating on different files, pipes, sockets, or when forking or execing in different processes. Biscuit uses read-lock-free lookups in some performance-critical code (see below).

Virtual Memory. Biscuit uses page-table hardware to implement zero-fill-on-demand memory allocation, copy-on-write fork, and lazy mapping of files (e.g., for `exec`) in which the PTEs are populated only when the process page-faults, and `mmap`.

Biscuit records contiguous memory mappings compactly, so in the common case large numbers of mapping objects aren't needed. Physical pages can have multiple references; Biscuit tracks these using reference counts.

File System. Biscuit implements a file system supporting the core POSIX file system calls. The file system has a file name lookup cache, a vnode cache, and a block cache. Biscuit guards each vnode with a mutex and resolves pathnames by first attempting each lookup in a read-lock-free directory cache before falling back to locking each directory named in the path, one after the other. Biscuit runs each file system call as a transaction and has a journal to commit updates to disk atomically. The journal batches transactions through deferred group commit, and allows file content writes to bypass the journal. Biscuit has an AHCI disk driver that uses DMA, command coalescing, native command queuing, and MSI interrupts.

Network Stack. Biscuit implements a TCP/IP stack and a driver for Intel PCI-Express Ethernet NICs in Go. The driver uses DMA and MSI interrupts. The system-call API provides POSIX sockets.

Limitations. Although Biscuit can run many Linux C programs without source modification, it is a research prototype and lacks many features. Biscuit does not support scheduling priority because it relies on the Go runtime scheduler. Biscuit is optimized for a small number of cores, but not for large multicore machines or NUMA. Biscuit does not swap or page out to disk, and does not implement the reverse mapping that would be required to steal mapped pages. Biscuit lacks many security features like users, access control lists, or address space randomization.

5 Garbage collection

Biscuit's use of garbage collection is a clear threat to its performance. This section outlines the Go collector's design and describes how Biscuit configures the collector; §8 evaluates performance costs.

5.1 Go's collector

Go 1.10 has a concurrent parallel mark-and-sweep garbage collector [21]. The concurrent aspect is critical for Biscuit, since it minimizes the collector's "stop-the-world" pauses.

When the Go collector is idle, the runtime allocates from the free lists built by the last collection. When the free space falls below a threshold, the runtime enables concurrent collection. When collection is enabled, the work of following ("tracing") pointers to find and mark reachable ("live") objects is interleaved with execution: each allocator call does a small amount of tracing and marking. Writes to already-traced objects are detected with compiler-generated "write barriers" so that any newly installed pointers will be traced. Once all pointers have been traced, the collector turns off write barriers and resumes ordinary execution. The collector suspends ordinary execution on all cores (a "stop-the-world" pause) twice during a collection: at the beginning to enable the write barrier on all cores and at the end to check that all objects have been marked. These stop-the-world pauses typically last dozens of microseconds. The collector rebuilds the free lists from the unmarked parts of memory ("sweeps"), again interleaved with Biscuit execution, and then becomes idle when all free heap memory has been swept. The collector does not move objects, so it does not reduce fragmentation.

The fraction of CPU time spent collecting is roughly proportional to the number of live objects, and inversely proportional to the interval between collections [20, 52]. This interval can be made large by devoting enough RAM to the heap that a substantial amount of space ("headroom") is freed by each collection.

The Go collector does most of its work during calls to the heap allocator, spreading out this work roughly evenly among calls. Thus goroutines see delays proportional to the amount that they allocate; §8.5 presents measurements of these delays for Biscuit.

5.2 Biscuit's heap size

At boot time, Biscuit allocates a fixed amount of RAM for its Go heap, defaulting to $1/32$nd of total RAM. Go's collector ordinarily expands the heap memory when live data exceeds half the existing heap memory; Biscuit disables this expansion. The next section (§6) explains how Biscuit copes with situations where the heap space is nearly filled with live data.

6 Avoiding heap exhaustion

Biscuit must address the possibility of live kernel data completely filling the RAM allocated for the heap ("heap exhaustion"). This is a difficult area that existing kernels struggle with (§2).

6.1 Approach: reservations

Biscuit is designed to tolerate heap exhaustion without kernel failure. In addition, it can take corrective action when there are identifiable "bad citizen" processes that allocate excessive kernel resources implemented with heap objects, such as the structures describing open files and pipes. Biscuit tries to identify bad citizens and kill them, in order to free kernel heap space and allow good citizens to make progress.

Biscuit's approach to kernel heap exhaustion has three elements. First, it purges caches and soft state as the heap nears exhaustion. Second, code at the start of each system call waits until it can reserve enough heap space to complete the call; the reservation ensures that the heap allocations made in the call will succeed once the wait (if any) is over. Third, a kernel "killer" thread watches for processes that are consuming lots of kernel heap when the heap is near exhaustion, and kills them.

This approach has some good properties. Applications do not have to cope with system call failures due to kernel heap exhaustion. Kernel code does not see heap allocation failure (with a few exceptions), and need not include logic to recover from such failures midway through a system call. System calls may have to wait for reservations, but they wait at their entry points without locks held, so deadlock is avoided.

The killer thread must distinguish between good and bad citizens, since killing a critical process (e.g., `init`) can make the system unusable. If there is no obvious "bad citizen," this approach may block and/or kill valuable processes. Lack of a way within POSIX for the kernel to gracefully revoke resources causes there to be no good solution in some out-of-memory situations.

The mechanisms in this section do not apply to non-heap allocations. In particular, Biscuit allocates physical memory pages from a separate allocator, not from the Go heap; page allocations can fail, and kernel code must check for failure and recover (typically by returning an error to a system call).

6.2 How Biscuit reserves

Biscuit dedicates a fixed amount of RAM M for the kernel heap. A system call only starts if it can reserve enough heap memory for the maximum amount of *simultaneously* live data that it uses, called s. A system call may allocate more than s from the heap, but the amount over s must be dead and can be freed by the collector. This means that, even in the extreme case in which all but s of the

```
reserve(s):
    g := last GC live bytes
    c := used bytes
    n := reserved bytes
    L := g + c + n
    M := heap RAM bytes
    if L + s < M:
        reserved bytes += s
    else:
        wake killer thread
        wait for OK from killer thread

release(s):
    a := bytes allocated by syscall
    if a < s:
        used bytes += a
    else:
        used bytes += s
    reserved bytes -= s
```

Figure 2: Pseudo code for heap reservations in Biscuit.

heap RAM is used by live data or is already reserved, the system call can execute, with collections as needed to recover the call's own dead data in excess of s.

Ideally, a reservation should check that M minus the amount of live and reserved data in the heap is greater than or equal to s. However, except immediately after a collection, the amount of live heap data is not known. Biscuit maintains a conservative over-estimate of live heap data using three counters: g, c, and n. g is the amount of live data marked by the previous garbage collection. c is the total amount of reservations made by system calls that have completed. n is the total outstanding reservations of system calls that are executing but not finished. Let L be the sum of g, c, and n.

Figure 2 presents pseudo code for reserving and releasing the reservation of heap RAM in Biscuit. Before starting a system call, a thread checks that $L+s < M$. If $L+s < M$, the thread reserves by adding s to n, otherwise the thread wakes up the killer thread and sleeps. When finished, a system call calculates a, the total amount actually allocated, and uses a to (partially) release any over-reservation: if $a <$s, the system call adds a to c and subtracts s from n. Otherwise, $a \geq s$ and the system call adds s to c and subtracts s from n.

The reason for separate c and n is to carry over reservations of system calls that span a garbage collection; a collection sets c to zero but leaves n unchanged.

If heap memory is plentiful (live data $\ll M$), the amount of live+dead data in the heap usually grows faster than L, so collections are triggered by heap free list exhaustion rather than by $L+s \geq M$. Thus system calls do not wait for memory, and do not trigger the killer thread. As live heap data increases, and $g+n$ gets close to M, $L+s$ may reach M before a collection would ordinarily be triggered. For this reason the killer thread performs a collection before deciding whether to kill processes.

6.3 Static analysis to find s

We have developed a tool, MAXLIVE, that analyzes the Biscuit source code and the Go packages Biscuit uses to find s for each system call. The core challenge is detecting statically when allocated memory can no longer be live, since many system calls allocate memory for transient uses. Other challenges include analyzing loops with non-constant bounds, and determining reservations for background kernel activities that are not associated with a specific system call.

We address these challenges by exploiting the characteristic event-handler-style structure of most kernel code, which does a modest amount of work and then returns (or goes idle); system call implementations, for example, work this way. Furthermore, we are willing to change the kernel code to make it amenable to the reservation approach, for example to avoid recursion (we changed a few functions). Two modifications were required to standard Go packages that Biscuit uses (packages *time* and *fmt*).

6.3.1 Basic MAXLIVE operation

MAXLIVE examines the call graph (using Go's *ssa* and *callgraph* packages) to detect all allocations a system call may perform. It uses escape and pointer analysis (Go's *pointer* package) to detect when an allocation does not "escape" above a certain point in the call graph, meaning that the allocation must be dead on return from that point.

MAXLIVE handles a few kinds of allocation specially: *go*, *defer*, maps, and slices. *go* (which creates a goroutine) is treated as an escaping allocation of the maximum kernel stack size (the new goroutine itself must reserve memory when it starts, much as if it were itself a system call). *defer* is a non-escaping allocation, but is not represented by an object in the SSA so MAXLIVE specifically considers it an allocation. Every insertion into a map or slice could double its allocated size; MAXLIVE generally doesn't know the old size, so it cannot predict how much memory would be allocated. To avoid this problem, we annotate the Biscuit source to declare the maximum size of slices and maps, which required 70 annotations.

6.3.2 Handling loops

For loops where MAXLIVE cannot determine a useful bound on the number of iterations, we supply a bound with an annotation; there were 78 such loops. Biscuit contains about 20 loops whose bounds cannot easily be expressed with an annotation, or for which the worst case is too large to be useful. Examples include retries to handle wakeup races in `poll`, iterating over a directory's data blocks during a path component lookup, and iterating over the pages of a user buffer in `write`.

We handle such loops with *deep reservations*. Each loop iteration tries to reserve enough heap for just the one

iteration. If there is insufficient free heap, the loop aborts and waits for free memory at the beginning of the system call, retrying when memory is available. Two loops (in `exec` and `rename`) needed code to undo changes after an allocation failure; the others did not.

Three system calls have particularly challenging loops: `exit`, `fork`, and `exec`. These calls can close many file descriptors, either directly or on error paths, and each close may end up updating the file system (e.g. on last close of a deleted file). The file system writes allocate memory, and may create entries in file system caches. Thus, for example, an exiting process that has many file descriptors may need a large amount of heap memory for the one `exit` system call. However, in fact `exit`'s memory requirements are much smaller than this: the cache entries will be deleted if heap memory is tight, so only enough memory is required to execute a single close. We bound the memory use of `close` by using MAXLIVE to find all allocations that may be live once `close` returns. We then manually ensure that all such allocations are either dead once `close` returns or are evictable cache entries. That way `exit`, `fork`, and `exec` only need to reserve enough kernel heap for one call to `close`. This results in heap bounds of less than 500kB for all system calls but `rename` and `fork` (1MB and 641kB, respectively). The `close` system call is the only one we manually analyze with the assistance of MAXLIVE.

6.3.3 Kernel threads

A final area of special treatment applies to long-running kernel threads. An example is the filesystem logging thread, which acts on behalf of many processes. Each long-running kernel thread has its own kernel heap reservation. Since `exit` must always be able to proceed when the killer thread kills a process, kernel threads upon which `exit` depends must never release their heap reservation. For example, `exit` may need to free the blocks of unlinked files when closing file descriptors and thus depends on the filesystem logging thread. Other kernel threads, like the ICMP packet processing thread, block and wait for heap reservations when needed and release them when idle.

6.3.4 Killer thread

The killer thread is woken up when a system call's reservation fails. The thread first starts a garbage collection and waits for it to complete. If the collection doesn't free enough memory, the killer thread asks each cache to free as many entries as possible, and collects again. If that doesn't yield enough free memory, the killer thread finds the process with the largest total number of mapped memory regions, file descriptors, and threads, in the assumption that it is a genuine bad citizen, kills it, and again collects. As soon as the killer thread sees that enough memory has been freed to satisfy the waiting reservation, it wakes up the waiting thread and goes back to sleep.

6.4 Limitations

Biscuit's approach for handling heap exhaustion requires that the garbage collector run successfully when there is little or no free memory available. However, Go's garbage collector may need to allocate memory during a collection in order to make progress, particularly for the work stack of outstanding pointers to scan. We haven't implemented it, but Biscuit could recover from this situation by detecting when the work stack is full and falling back to using the mark bitmap as the work stack, scanning for objects which are marked but contain unmarked pointers. This strategy will allow the garbage collection to complete, but will likely be slow. We expect this situation to be rare since the work stack buffers can be preallocated for little cost: in our experiments, the garbage collector allocates at most 0.8% of the heap RAM for work stacks.

Because the Go collector doesn't move objects, it doesn't reduce fragmentation. Hence, there might be enough free memory but in fragments too small to satisfy a large allocation. To eliminate this risk, MAXLIVE should compute s for each size class of objects allocated during a system call. Our current implementation doesn't do this yet.

6.5 Heap exhaustion summary

Biscuit borrows ideas for heap exhaustion from Linux: the killer thread, and the idea of waiting and retrying after the killer thread has produced free memory. Biscuit simplifies the situation by using reservation checks at the start of each system call, rather than Linux's failure checks at each allocation point; this means that Biscuit has less recovery code to back out of partial system calls, and can wait indefinitely for memory without fear of deadlock. Go's static analyzability helped automate Biscuit's simpler approach.

7 Implementation

The Biscuit kernel is written almost entirely in Go: Figure 3 shows that it has 27,583 lines of Go, 1,546 lines of assembly, and no C.

Biscuit provides 58 system calls, listed in Figure 4. It has enough POSIX compatibility to run some existing server programs (for example, NGINX and Redis).

Biscuit includes device drivers for AHCI SATA disk controllers and for Intel 82599-based Ethernet controllers such as the X540 10-gigabit NIC. Both drivers use DMA. The drivers use Go's `unsafe.Pointer` to access device registers and in-memory structures (such as DMA descriptors) defined by device hardware, and Go's `atomic` package to control the order of these accesses. The code

Component	Lang	LOC
Biscuit kernel (mostly boot)	asm	546
Biscuit kernel	Go	
Core		1700
Device drivers		4088
File system		7338
Network		4519
Other		1105
Processes		935
Reservations		749
Syscalls		5292
Virtual memory		1857
Total		27583
MaxLive	Go	1299
Runtime modifications	asm	1,000
Runtime modifications	Go	3,200

Figure 3: Lines of code in Biscuit. Not shown are about 50,000 lines of Go runtime and 32,000 lines of standard Go packages that Biscuit uses.

would be more concise if Go supported some kind of memory fence.

Biscuit contains 90 uses of Go's "unsafe" routines (excluding uses in the Go runtime). These unsafe accesses parse and format packets, convert between physical page numbers and pointers, read and write user memory, and access hardware registers.

We modified the Go runtime to record the number of bytes allocated by each goroutine (for heap reservations), to check for runnable device handler goroutines, and to increase the default stack size from 2kB to 8kB to avoid stack expansion for a few common system calls.

Biscuit lives with some properties of the Go runtime and compiler in order to avoid significantly modifying them. The runtime does not turn interrupts off when holding locks or when manipulating a goroutine's own private state. Therefore, in order to avoid deadlock, Biscuit interrupt handlers just set a flag indicating that a device handler goroutine should wake up. Biscuit's timer interrupt handler cannot directly force goroutine context switches because the runtime might itself be in the middle of a context switch. Instead, Biscuit relies on Go's pre-emption mechanism for kernel goroutines (the Go compiler inserts pre-emption checks in the generated code). Timer interrupts do force context switches when they arrive from user space.

Goroutine scheduling decisions and the context switch implementation live in the runtime, not in Biscuit. One consequence is that Biscuit does not control scheduling policy; it inherits the runtime's policy. Another consequence is that per-process page tables are not switched when switching goroutines, so Biscuit system call code cannot safely dereference user addresses directly. Instead, Biscuit explicitly translates user virtual addresses to physical addresses, and also explicitly checks page permissions. Biscuit switches page tables if necessary before switching to user space.

We modified the runtime in three ways to reduce delays due to garbage collection. First, we disabled the dedicated garbage collector worker threads so that application threads don't compete with garbage collector threads for CPU cycles. Second, we made root marking provide allocation credit so that an unlucky allocating thread wouldn't mark many roots all at once. Third, we reduced the size of the pieces that large objects are broken into for marking from 128kB to 10kB.

Biscuit implements many standard kernel performance optimizations. For example, Biscuit maps the kernel text using large pages to reduce iTLB misses, uses per-CPU NIC transmit queues, and uses read-lock-free data structures in some performance critical code such as the directory cache and TCP polling. In general, we found that Go did not hinder optimizations.

8 Evaluation

This section analyzes the costs and benefits of writing a kernel in an HLL by exploring the following questions:

- To what degree does Biscuit benefit from Go's high-level language features? To answer, we count and explain Biscuit's use of these features (§8.1).
- Do C kernels have safety bugs that a high-level language might mitigate? We evaluate whether bugs reported in Linux kernel CVEs would likely apply to Biscuit (§8.2).
- How much performance does Biscuit pay for Go's HLL features? We measure the time Biscuit spends in garbage collection, bounds checking, etc., and the delays that GC introduces (§8.4,8.5,8.6).
- What is the performance impact of using Go instead of C? We compare nearly-identical pipe and page-fault handler implementations in Go and C (§8.7).
- Is Biscuit's performance in the same ballpark as Linux, a C kernel (§8.8)?
- Is Biscuit's reservation scheme effective at handling kernel heap exhaustion (§8.9)?
- Can Biscuit benefit from RCU-like lock-free lookups (§8.10)?

8.1 Biscuit's use of HLL features

Our subjective feeling is that Go has helped us produce clear code and helped reduce programming difficulty, primarily by abstracting and automating low-level tasks.

Figure 5 shows how often Biscuit uses Go's HLL features, and compares with two other major Go systems:

accept	bind	chdir	close	connect	dup2	execv	exit
fcntl	fork	fstat	ftruncate	futex	getcwd	getpid	getppid
getrlimit	getrusage	getsockopt	gettid	gettimeofday	info	kill	link
listen	lseek	mkdir	mknod	mmap	munmap	nanosleep	open
pipe2	poll	pread	prof	pwrite	read	readv	reboot
recvfrom	recvmsg	rename	sendmsg	sendto	setrlimit	setsockopt	shutdown
socket	socketpair	stat	sync	threxit	truncate	unlink	wait4
write	writev						

Figure 4: Biscuit's 58 system calls.

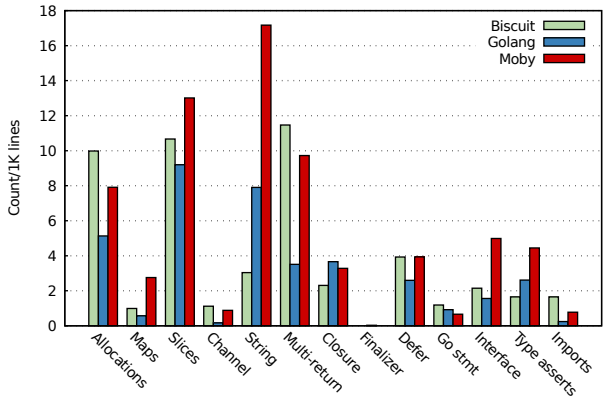


Figure 5: Uses of Go HLL features in the Git repositories for Biscuit, Golang (1,140,318 lines), and Moby (1,004,300 lines) per 1,000 lines. For data types (such as slices), the numbers indicate the number of declarations of a variable, argument, or structure field of that type.

the Golang repository (containing Go's compiler, runtime, and standard packages), and Moby[1], which contains Docker's container software and is the most starred Go repository on Github at the time of writing. Figure 5 shows the number of times each feature is used per 1,000 lines of code. Biscuit uses Go's HLL features about as much as other Go systems software.

To give a sense how these HLL features can benefit a kernel, the rest of this section provides examples of successful uses, as well as situations where we didn't use them. Biscuit relies on the Go allocator and garbage collector for nearly all kernel objects. Biscuit has 302 statements that allocate an object from the GC-managed heap. Some of the objects are compound (composed of multiple Go objects). For example, Biscuit's `Vmregion_t`, which describes a mapped region of virtual memory, has a red/black tree of `Vminfo_t`, which itself is compound (e.g., when it is backed by a file). The garbage collector eliminates the need for explicit code to free the parts of such compound data types.

Biscuit's only special-purpose allocator is its physical page allocator. It is used for process memory pages, file cache pages, socket and pipe buffers, and page table pages.

[1] `https://github.com/moby/moby`

Biscuit uses many goroutines. For example, device drivers create long-running goroutines to handle events such as packet arrival. Biscuit avoids goroutine creation, however, in frequently executed code. The reason is that the garbage collector produces pauses proportional to the number of goroutines; these are insignificant for thousands of goroutines but a problem with hundreds of thousands.

The combination of threads and garbage collection is particularly pleasing, since it avoids forcing the programmer to worry about delaying frees for shared objects until the last sharing thread has finished. For example, Biscuit's `poll` system call installs a pointer to a helper object in each file descriptor being polled. When input arrives on a descriptor, the goroutine delivering the input uses the helper object to wake up the polling thread. Garbage collection eliminates races between arriving input and freeing the helper object.

Some Biscuit objects, when the last reference to them disappears, need to take clean-up actions before their memory can be collected; for example, TCP connections must run the TCP shutdown protocol. Go's finalizers were not convenient in these situations because of the prohibition against cycles among objects with finalizers. Biscuit maintains reference counts in objects that require clean-up actions.

Biscuit uses many standard Go packages. For example, Biscuit imports `sync` in 28 files and `atomic` packages in 18 files. These packages provide mutexes, condition variables, and low-level atomic memory primitives. Biscuit's MAXLIVE tool depends on Go's code analysis packages (`ssa`, `callgraph`, and `pointer`).

Biscuit itself is split into 31 Go packages. Packages allowed some code to be developed and tested in user space. For example, we tested the file system package for races and crash-safety in user space. The package system also made it easy to use the file system code to create boot disks.

8.2 Potential to reduce bugs

An HLL might help avoid problems such as memory corruption from buffer overflows. To see how this applies to kernels, we looked at Linux execute-code bugs in the CVE database published in 2017 [34]. There are 65 bugs

Type	CVE-...
Use-after-free or double-free	2016-10290, 2016-10288, 2016-8480, 2016-8449, 2016-8436, 2016-8392, 2016-8391, 2016-6791
Out-of-bounds access	2017-1000251, 2017-6264, 2017-0622, 2017-0621, 2017-0620, 2017-0619, 2017-0614, 2017-0613, 2017-0612, 2017-0611, 2017-0608, 2017-0607, 2017-0521, 2017-0520, 2017-0465, 2017-0458, 2017-0453, 2017-0443, 2017-0442, 2017-0441, 2017-0440, 2017-0439, 2017-0438, 2017-0437, 2016-10289, 2016-10285, 2016-10283, 2016-8476, 2016-8421, 2016-8420, 2016-8419, 2016-6755

Figure 6: Linux kernel CVEs from 2017 that would *not* cause memory corruption, code execution, or information disclosure in Biscuit.

where the patch is publicly available. For 11 bugs of the 65, we aren't sure whether Go would have improved the outcome. 14 of the 65 are logic bugs that could arise as easily in Go as they do in C. Use of Go would have improved the outcome of the remaining 40 bugs (listed in Figure 6), based on manual inspection of the patch that fixed the bug. The impact of some of these 40 bugs is severe: several allow remote code execution or information disclosure. Many of the bugs in the out-of-bounds category would have resulted in runtime errors in Go, and caused a panic. This is not ideal, but better than allowing a code execution or information disclosure exploit. The bugs in the use-after-free category would not have occurred in Go, because garbage collection would obviate them.

The Go runtime and packages that Biscuit relies on also have bugs. There are 14 CVEs in Go published from 2016 to 2018. Two of them allow code execution (all in `go get`) and two allow information gain (due to bugs in Go's *smtp* and *math/big* packages).

8.3 Experimental Setup

The performance experiments reported below were run on a four-core 2.8 GHz Xeon X3460 with hyper-threading disabled and 16 GB of memory. Biscuit uses Go version 1.10. Except where noted, the benchmarks use an in-memory file system, rather than a disk, in order to stress the CPU efficiency of the kernel. The in-memory file system is the same as the disk file system, except that it doesn't append disk blocks to the in-memory log or call the disk driver. The disk file system uses a Samsung 850 SSD.

The network server benchmarks have a dedicated ten-gigabit Ethernet switch between a client and a server machine, with no other traffic. The machines use Intel X540 ten-gigabit network interfaces. The network interfaces use an interrupt coalescing period of 128 μs. The client runs Linux.

Except when noted, Biscuit allocates 512MB of RAM to the kernel heap. The reported fraction of CPU time spent in the garbage collector is calculated as $\frac{O_{gc}-O_{nogc}}{O_{gc}}$, where O_{gc} is the time to execute a benchmark with garbage collection and O_{nogc} is the time without garbage collection. To measure O_{nogc}, we reserve enough RAM for the kernel heap that the kernel doesn't run out of free memory and thus never collects. This method does not remove the cost to check, for each write, whether write barriers are enabled.

We report the average of three runs for all figures except maximums. Except when noted, each run lasts for one minute, and variation in repeated runs for all measurements is less than 3%.

Many of the performance experiments use three applications, all of which are kernel-intensive:

CMailbench CMailbench is a mail-server-like benchmark which stresses the virtual memory system via `fork` and `exec`. The benchmark runs four server processes and four associated clients, all on the same machine. For each message delivery, the client forks and execs a helper; the helper sends a 1660-byte message to its server over a UNIX-domain socket; the server forks and execs a delivery agent; the delivery agent writes the message to a new file in a separate directory for each server. Each message involves two calls to each of `fork`, `exec`, and `rename` as well as one or two calls to `read`, `write`, `open`, `close`, `fstat`, `unlink`, and `stat`.

NGINX NGINX [38] (version 1.11.5) is a high-performance web server. The server is configured with four processes, all of which listen on the same socket for TCP connections from clients. The server processes use `poll` to wait for input on multiple connections. NGINX's request log is disabled. A separate client machine keeps 64 requests in flight; each request involves a fresh TCP connection to the server. For each incoming connection, a server process parses the request, opens and reads a 612-byte file, sends the 612 bytes plus headers to the client, and closes the connection. All requests fetch the same file.

Redis Redis (version 3.0.5) is an in-memory key/value database. We modified it to use `poll` instead of `select` (since Biscuit doesn't support `select`). The benchmark runs four single-threaded Redis server processes. A client machine generates load over the network using two instances of Redis's "redis-benchmark" per Redis server process, each of which opens 100 connections to the Redis process and keeps a single GET outstanding on each connection. Each GET requests one of 10,000 keys at random. The values are two bytes.

8.4 HLL tax

This section investigates the performance costs of Go's HLL features for the three applications. Figure 7 shows the results.

The "Tput" column shows throughput in application requests per second.

The "Kernel time" column (fraction of time spent in the kernel, rather than in user space) shows that the results are dominated by kernel activity. All of the benchmarks keep all four cores 100% busy.

The applications cause Biscuit to average between 18 and 48 MB of live data in the kernel heap. They allocate transient objects fast enough to trigger dozens of collections during each benchmark run ("GCs"). These collections use between 1% and 3% of the total CPU time.

"Prologue cycles" are the fraction of total time used by compiler-generated code at the start of each function that checks whether the stack must be expanded, and whether the garbage collector needs a stop-the-world pause. "WB cycles" reflect compiler-generated write-barriers that take special action when an object is modified during a concurrent garbage collection.

"Safety cycles" reports the cost of runtime checks for nil pointers, array and slice bounds, divide by zero, and incorrect dynamic casts. These checks occur throughout the compiler output; we wrote a tool that finds them in the Biscuit binary and cross-references them with CPU time profiles.

"Alloc cycles" measures the time spent in the Go allocator, examining free lists to satisfy allocation requests (but not including concurrent collection work). Allocation is not an HLL-specific task, but it is one that some C kernels streamline with custom allocators [8].

Figure 7 shows that the function prologues are the most expensive HLL feature. Garbage collection costs are noticeable but not the largest of the costs. On the other hand, §8.6 shows that collection cost grows with the amount of live data, and it seems likely that prologue costs could be reduced.

8.5 GC delays

We measured the delays caused by garbage collection (including interleaved concurrent work) during the execution of NGINX, aggregated by allocator call, system call, and NGINX request.

0.7% of heap allocator calls are delayed by collection work. Of the delayed allocator calls, the average delay is 0.9 microseconds, and the worst case is 115 microseconds, due to marking a large portion of the TCP connection hashtable.

2% of system calls are delayed by collection work; of the delayed system calls, the average delay is 1.5 microseconds, and the worst case is 574 microseconds, incurred by a *poll* system call that involved 25 allocator calls that performed collection work.

22% of NGINX web requests are delayed by collection work. Of the delayed requests, the average total collection delay is 1.8 microseconds (out of an average request processing time of 45 microseconds). Less than 0.3% of requests spend more than 100 microseconds garbage collecting. The worst case is 582 microseconds, which includes the worst-case system call described above.

8.6 Sensitivity to heap size

A potential problem with garbage collection is that it consumes a fraction of CPU time proportional to the "headroom ratio" between the amount of live data and the amount of RAM allocated to the heap. This section explores the effect of headroom on collection cost.

This experiment uses the CMailbench benchmark. We artificially increased the live data by inserting two or four million vnodes (640 or 1280 MB of live data) into Biscuit's vnode cache. We varied the amount of RAM allocated to the kernel heap.

Figure 8 shows the results. The two most significant columns are "Headroom ratio" and "GC%;" together they show roughly the expected relationship. For example, comparing the second and last table rows shows that increasing both live data and total heap RAM, so that the ratio remains the same, does not change the fraction of CPU time spent collecting; the reason is that the increased absolute amount of headroom decreases collection frequency, but that is offset by the fact that doubling the live data doubles the cost of each individual collection.

In summary, while the benchmarks in §8.4 / Figure 7 incur modest collection costs, a kernel heap with millions of live objects but limited heap RAM might spend a significant fraction of its time collecting. We expect that decisions about how much RAM to buy for busy machines would include a small multiple (2 or 3) of the expected peak kernel heap live data size.

8.7 Go versus C

This section compares the performance of code paths in C and Go that are nearly identical except for language. The goal is to focus on the impact of language choice on performance for kernel code. The benchmarks involve a small amount of code because of the need to ensure that the C and Go versions are very similar.

The code paths are embedded in Biscuit (for Go) and Linux (for C). We modified both to ensure that the kernel code paths exercised by the benchmarks are nearly identical. We disabled Linux's kernel page-table isolation, retpoline, address space randomization, transparent hugepages, hardened usercopy, cgroup, fair group, and bandwidth scheduling, scheduling statistics, ftrace, kprobes, and paravirtualization to make its code paths similar to Biscuit. We also disabled Linux's FS notifications, *atime* and *mtime* updates to pipes, and replaced Linux's

	Tput	Kernel time	Live data	GCs	GC cycles	Prologue cycles	WB cycles	Safety cycles	Alloc cycles
CMailbench	15,862	92%	34 MB	42	3%	6%	0.9%	3%	8%
NGINX	88,592	80%	48 MB	32	2%	6%	0.7%	2%	9%
Redis	711,792	79%	18 MB	30	1%	4%	0.2%	2%	7%

Figure 7: Measured costs of HLL features in Biscuit for three kernel-intensive benchmarks. "Alloc cycles" are not an HLL-specific cost, since C code has significant allocation costs as well.

Live (MB)	Total (MB)	Headroom ratio	Tput (msg/s)	GC%	GCs
640	960	0.66	10,448	34%	43
640	1280	0.50	12,848	19%	25
640	1920	0.33	14,430	9%	13
1280	2560	0.50	13,041	18%	12

Figure 8: CMailbench throughput on Biscuit with different kernel heap sizes. The columns indicate live heap memory; RAM allocated to the heap; the ratio of live heap memory to heap RAM; the benchmark's throughput on Biscuit; the fraction of CPU cycles (over all four cores) spent garbage collecting; and the number of collections.

scheduler and page allocator with simple versions, like Biscuit's. The benchmarks allocate no heap memory in steady-state, so Biscuit's garbage collector is not invoked.

8.7.1 Ping-pong

The first benchmark is "ping-pong" over a pair of pipes between two user processes. Each process takes turns performing five-byte reads and writes to the other process. Both processes are pinned to the same CPU in order to require the kernel to context switch between them. The benchmark exercises core kernel tasks: system calls, sleep/wakeup, and context switch.

We manually verified the similarity of the steady-state kernel code paths (1,200 lines for Go, 1,786 lines for C, including many comments and macros which compile to nothing). The CPU-time profiles for the two showed that time was spent in near-identical ways. The ten most expensive instructions match: saving and restoring SSE registers on context switch, entering and exiting the kernel, *wrmsr* to restore the thread-local-storage register, the copy to/from user memory, atomic instructions for locks, and *swapgs*.

The results are 465,811 round-trips/second for Go and 536,193/second for C; thus C is 15% faster than Go on this benchmark. The benchmark spends 91% and 93% of its time in the kernel (as opposed to user space) for Go and C, respectively. A round trip takes 5,259 instructions for Go and 4,540 for C. Most of the difference is due to HLL features: 250, 200, 144, and 112 instructions per round-trip for stack expansion prologues, write barrier, bounds, and nil pointer/type checks, respectively.

	Biscuit	Linux	Ratio
CMailbench (mem)	15,862	17,034	1.07
CMailbench (SSD)	254	252	0.99
NGINX	88,592	94,492	1.07
Redis	711,792	775,317	1.09

Figure 9: Application throughput of Biscuit and Linux. "Ratio" is the Linux to Biscuit throughput ratio.

8.7.2 Page-faults

The second Go-versus-C benchmark is a user-space program that repeatedly calls `mmap()` to map 4 MB of zero-fill-on-demand 4096-byte pages, writes a byte on each page, and then unmaps the memory. Both kernels initially map the pages lazily, so that each write generates a page fault, in which the kernel allocates a physical page, zeroes it, adds it to the process page table, and returns. We ran the benchmark on a single CPU on Biscuit and Linux and recorded the average number of page-faults per second.

We manually verified the similarity of the steady-state kernel code: there are about 480 and 650 lines of code for Biscuit and Linux, respectively. The benchmark spends nearly the same amount of time in the kernel on both kernels (85% on Biscuit and 84% on Linux). We verified with CPU-time profiles that the top five most expensive instructions match: entering the kernel on the page-fault, zeroing the newly allocated page, the userspace store after handling the fault, saving registers, and atomics for locks.

The results are 731 nanoseconds per page-fault for Go and 695 nanoseconds for C; C is 5% faster on this benchmark. The two implementations spend much of their time in three ways: entering the kernel's page-fault handler, zeroing the newly allocated page, and returning to userspace. These operations use 21%, 22%, and 15% of CPU cycles for Biscuit and 21%, 20%, and 16% of CPU cycles for Linux, respectively.

These results give a feel for performance differences due just to choice of language. They don't involve garbage collection; for that, see §8.4 and §8.6.

8.8 Biscuit versus Linux

To get a sense of whether Biscuit's performance is in the same ballpark as a high-performance C kernel, we report the performance of Linux on the three applications of §8.4. The applications make the same system calls on Linux and on Biscuit. These results cannot be used to conclude

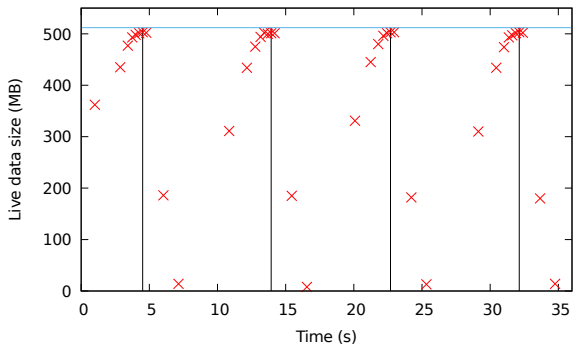


Figure 10: The amount of live data (in red) in the kernel heap during the first 35 seconds of the heap exhaustion experiment. The blue line indicates the RAM allocated to the kernel heap (512MB). The four vertical black lines indicate the points at which the killer thread killed the abusive child process.

much about performance differences due to Biscuit's use of Go, since Linux includes many features that Biscuit omits, and Linux may sacrifice some performance on these benchmarks in return for better performance in other situations (e.g., large core counts or NUMA).

We use Debian 9.4 with Linux kernel 4.9.82. We increased Linux's performance by disabling some costly features: kernel page-table isolation, retpoline, address space randomization, transparent hugepages, TCP selective ACKs, and SYN cookies. We replaced *glibc* with *musl* (nearly doubling the performance of CMailbench on Linux) and pinned the application threads to CPUs when it improves the benchmark's performance. We ran CMailbench in two configurations: one using an in-memory file system and the other using an SSD file system (*tmpfs* and *ext-4* on Linux, respectively). The benchmarks use 100% of all cores on both Biscuit and Linux, except for CMailbench (SSD), which is bottlenecked by the SSD. The proportion of time each benchmark spends in the kernel on Linux is nearly the same as on Biscuit (differing by at most two percentage points).

Figure 9 presents the results: Linux achieves up to 10% better performance than Biscuit. The "HLL taxes" identified in §8.4 contribute to the results, but the difference in performance is most likely due to the fact that the two kernels have different designs and amounts of functionality. It took effort to make Biscuit achieve this level of performance. Most of the work was in understanding why Linux was more efficient than Biscuit, and then implementing similar optimizations in Biscuit. These optimizations had little to do with the choice of language, but were for the most part standard kernel optimizations (e.g., avoiding lock contention, avoiding TLB flushes, using better data structures, adding caches).

8.9 Handling kernel heap exhaustion

This experiment demonstrates two things. First, that the system calls of a good citizen process do not fail when executing concurrently with an application that tries to exhaust the kernel heap. Second, that Biscuit's heap RAM reservations aren't too conservative: that the reservations allow most of the heap RAM to be used before forcing system calls to wait.

The experiment involves two programs. An abusive program repeatedly forks a child and waits for it. The child creates many non-contiguous memory mappings, which cause the kernel to allocate many heap objects describing the mappings. These objects eventually cause the kernel heap to approach fullness, at which point the out-of-memory killer kills the child. Meanwhile, a well-behaved program behaves like a UNIX mail daemon: it repeatedly delivers dozens of messages and then sleeps for a few seconds. This process complains and exits if any of its system calls returns an unexpected error. The kernel has 512MB of RAM allocated to its heap. The programs run for 25 minutes, and we record the amount of live data in the kernel heap at the end of every garbage collection.

Figure 10 shows the first 35 seconds of the experiment. Each red cross indicates the amount of live kernel heap data after a GC. The blue line at the top corresponds to 512MB. The four vertical lines show the times at which the out-of-memory killer killed the abusive program's child process.

Biscuit allows the live data in its heap to grow to about 500 MB, or 97% of the heap RAM. The main reason that live data does not reach 512 MB is that the reservation for the file system logger thread is 6 MB, more than the thread actually uses. When the child is killed, it takes a couple seconds to release the kernel heap objects describing its many virtual memory mappings. The system calls of the good citizen process wait for reservations hundreds of thousands of times, but none return an error.

8.10 Lock-free lookups

This section explores whether read-lock-free data structures in Go increase parallel performance.

C kernels often use read-lock-free data structures to increase performance when multiple cores read the data. The goal is to allow reads without locking or dirtying cache lines, both of which are expensive when there is contention. However, safely deleting objects from a data structure with lock-free readers requires the deleter to defer freeing memory that a thread might still be reading. Linux uses read-copy update (RCU) to delay such frees, typically until all cores have performed a thread context switch; coupled with a rule that readers not hold references across context switch, this ensures safety [32, 33]. Linux's full set of RCU rules is complex; see "Review Checklist for RCU patches" [31].

Directory cache	Tput
Lock-free lookups	15,862 msg/s
Read-locked lookups	14,259 msg/s

Figure 11: The performance of CMailbench with two versions of Biscuit's directory cache, one read-lock-free and one using read locks.

Garbage collection automates the freeing decision, simplifying use of read-lock-free data structures and increasing the set of situations in which they can safely be used (e.g. across context switches). However, HLLs and garbage collection add their own overheads, so it is worth exploring whether read-lock-free data structures nevertheless increase performance.

In order to explore this question, we wrote two variants of a directory cache for Biscuit, one that is read-lock-free and one with read-locks. Both versions use an array of buckets as a hash table, each bucket containing a singly-linked list of elements. Insert and delete lock the relevant bucket, create new versions of list elements to be inserted or updated, and modify next pointers to refer to the new elements. The read-lock-free version of lookup simply traverses the linked list.[2] The read-locked version first read-locks the bucket (forbidding writers but allowing other readers) and then traverses the list. We use CMailbench for the benchmark since it stresses creation and deletion of entries in the directory cache. The file system is in-memory, so there is no disk I/O.

Figure 11 shows the throughput of CMailbench using the read-lock-free directory cache and the read-locked directory cache. The read-lock-free version provides an 11% throughput increase: use of Go does not eliminate the performance advantage of read-lock-free data in this example.

9 Discussion and future work

Should one write a kernel in Go or in C? We have no simple answer, but we can make a few observations. For existing large kernels in C, the programming cost of conversion to Go would likely outweigh the benefits, particularly considering investment in expertise, ecosystem, and development process. The question makes more sense for new kernels and similar projects such as VMMs.

If a primary goal is avoiding common security pitfalls, then Go helps by avoiding some classes of security bugs (see §8.2). If the goal is to experiment with OS ideas, then Go's HLL features may help rapid exploration of different designs (see §8.1). If CPU performance is paramount, then C is the right answer, since it is faster (§8.4, §8.5). If efficient memory use is vital, then C is also the right answer: Go's garbage collector needs a factor of 2 to 3 of heap headroom to run efficiently (see §8.6).

We have found Go effective and pleasant for kernel development. Biscuit's performance on OS-intensive applications is good (about 90% as fast as Linux). Achieving this performance usually involved implementing the right optimizations; Go versus C was rarely an issue.

An HLL other than Go might change these considerations. A language without a compiler as good as Go's, or whose design was more removed from the underlying machine, might perform less well. On the other hand, a language such as Rust that avoids garbage collection might provide higher performance as well as safety, though perhaps at some cost in programmability for threaded code.

There are some Biscuit-specific issues we would like to explore further. We would like Biscuit to expand and contract the RAM used for the heap dynamically. We would like to modify the Go runtime to allow Biscuit to control scheduling policies. We would like to scale Biscuit to larger numbers of cores. Finally, we would like to explore if Biscuit's heap reservation scheme could simplify the implementation of C kernels.

10 Conclusions

Our subjective experience using Go to implement the Biscuit kernel has been positive. Go's high-level language features are helpful in the context of a kernel. Examination of historical Linux kernel bugs due to C suggests that a type- and memory-safe language such as Go might avoid real-world bugs, or handle them more cleanly than C. The ability to statically analyze Go helped us implement defenses against kernel heap exhaustion, a traditionally difficult task.

The paper presents measurements of some of the performance costs of Biscuit's use of Go's HLL features, on a set of kernel-intensive benchmarks. The fraction of CPU time consumed by garbage collection and safety checks is less than 15%. The paper compares the performance of equivalent kernel code paths written in C and Go, finding that the C version is about 15% faster.

We hope that this paper helps readers to make a decision about whether to write a new kernel in C or in an HLL.

Acknowledgements

We thank Nickolai Zeldovich, PDOS, Austin Clements, the anonymous reviewers, and our shepherd, Liuba Shrira, for their feedback. This research was supported by NSF award CSR-1617487.

[2] We used Go's atomic package to prevent re-ordering of memory reads and writes; it is not clear that this approach is portable.

References

[1] A. Anagnostopoulos. gopher-os, 2018. https://github.com/achilleasa/gopher-os.

[2] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, Sept. 2010.

[3] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *In Proceedings of the 2000 Usenix Annual Technical Conference*, pages 197–210. USENIX Association, 2000.

[5] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.

[6] F. J. Ballesteros. The Clive operating system, 2014. http://lsub.org/ls/clive.html.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[8] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer Conference*, 1994.

[9] J. Corbet. The too small to fail memory-allocation rule. from https://lwn.net/Articles/627419/, Dec 2014.

[10] J. Corbet. Revisiting too small to fail. from https://lwn.net/Articles/723317/, May 2017.

[11] D Language Foundation. D programming language, 2017. https://dlang.org/.

[12] D. Evans. cs4414: Operating Systems, 2014. http://www.rust-class.org/.

[13] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[14] C. M. Geschke, J. H. Morris, Jr., and E. H. Satterthwaite. Early experience with Mesa. *SIGOPS Oper. Syst. Rev.*, Apr. 1977.

[15] Google. The Go Programming Language, 2017. https://golang.org/.

[16] Google. gvisor, 2018. https://github.com/google/gvisor.

[17] R. D. Greenblatt, T. F. Knight, J. T. Holloway, and D. A. Moon. A LISP machine. In *Proceedings of the Fifth Workshop on Computer Architecture for Non-numeric Processing*, CAW '80, pages 137–138, New York, NY, USA, 1980. ACM.

[18] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[19] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, 1998.

[20] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM OOPSLA*, 2005.

[21] R. Hudson. Go GC: Prioritizing low latency and simplicity. from https://blog.golang.org/go15gc, Aug 2015.

[22] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 37–49, Stevenson, WA, Oct. 2007.

[23] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, Oct. 2005.

[24] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: A Wait-free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, Beijing, China, 2012. ACM.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect

of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[26] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26, New York, NY, USA, 2015. ACM.

[27] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM.

[28] A. Light. Reenix: implementing a Unix-like operating system in Rust, Apr. 2015.

[29] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, Houston, TX, Mar. 2013.

[30] B. McCloskey, D. F. Bacon, P. Cheng, and D. Grove. Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors. Technical report, IBM, 2008.

[31] P. McKenney. Review list for RCU patches. `https://www.kernel.org/doc/Documentation/RCU/checklist.txt`.

[32] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU usage in the Linux kernel: One decade later. 2012.

[33] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[34] MITRE Corporation. CVE Linux Kernel Vulnerability Statistics, 2018. `http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`.

[35] J. Mogul. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX 1996 Annual Technical Conference*, January 1996.

[36] Mozilla research. The Rust Programming Language, 2017. `https://doc.rust-lang.org/book/`.

[37] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[38] NGINX. Nginx, 2018. `https://www.nginx.com/`.

[39] P. Oppermann. Writing an OS in Rust, 2017. `http://os.phil-opp.com/`.

[40] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[41] W. M. Petullo, W. Fei, J. A. Solworth, and P. Gavlin. Ethos' deeply integrated distributed types. In *IEEE Security and Privacy Workshop on LangSec*, May 2014.

[42] D. Picheta. Nim in action, 2017. `http://nim-lang.org/`.

[43] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise Garbage Collection for C. In *Proceedings of the 9th International Symposium on Memory Management*, ISMM '09, Dublin, Ireland, June 2009. ACM.

[44] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, 1979. ACM.

[45] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 83–90, New York, NY, USA, 1989. ACM.

[46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, 2012. USENIX.

[47] A. S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, 2008.

[48] W. Teitelman. The Cedar programming environment: A midterm report and examination. Technical Report CSL-83-11, Xerox PARC, 1984.

[49] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Apr. 1987.

[50] Ticki. Redox - Your Next(Gen) OS, 2017. `https://doc.redox-os.org/book/`.

[51] L. Torvalds. `http://harmful.cat-v.org/software/c++/linus`, Jan 2004.

[52] T. Yang, M. Hertz, E. Berger, S. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *ACM ISMM*, 2004.

Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS

Ahmed Khawaja[1], Joshua Landgraf[1], Rohith Prakash[1],
Michael Wei[2], Eric Schkufza[2], Christopher J. Rossbach[3]
[1]*The University of Texas at Austin* [2]*VMware Research Group*
[3]*The University of Texas at Austin and VMware Research Group*

Abstract

Cloud providers such as Amazon and Microsoft have begun to support on-demand FPGA acceleration in the cloud, and hardware vendors will support FPGAs in future processors. At the same time, technology advancements such as 3D stacking, through-silicon vias (TSVs), and FinFETs have greatly increased FPGA density. The massive parallelism of current FPGAs can support not only extremely large applications, but multiple applications simultaneously as well.

System support for FPGAs, however, is in its infancy. Unlike software, where resource configurations are limited to simple dimensions of compute, memory, and I/O, FPGAs provide a multi-dimensional sea of resources known as the FPGA *fabric*: logic cells, floating point units, memories, and I/O can all be wired together, leading to spatial constraints on FPGA resources. Current stacks either support only a single application or statically partition the FPGA fabric into fixed-size *slots*. These designs cannot efficiently support diverse workloads: the size of the largest slot places an artificial limit on application size, and oversized slots result in wasted FPGA resources and reduced concurrency.

This paper presents AMORPHOS, which encapsulates user FPGA logic in *morphable tasks*, or **Morphlets**. Morphlets provide isolation and protection across mutually distrustful protection domains, extending the guarantees of software processes. Morphlets can *morph*, dynamically altering their deployed form based on resource requirements and availability. To build Morphlets, developers provide a parameterized hardware design that interfaces with AMORPHOS, along with a *mesh*, which specifies external resource requirements. AMORPHOS explores the parameter space, generating deployable *Morphlets* of varying size and resource requirements. AMORPHOS multiplexes Morphlets on the FPGA in both space and time to maximize FPGA utilization.

We implement AMORPHOS on Amazon F1 [1] and Microsoft Catapult [92]. We show that protected sharing and dynamic scalability support on workloads such as DNN inference and blockchain mining improves aggregate throughput up to 4× and 23× on Catapult and F1 respectively.

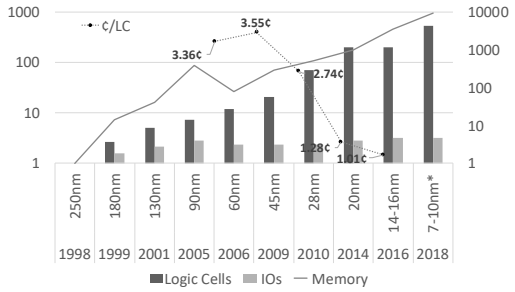


Figure 1: Cost per logic cell and relative density of memory and logic cells over time for FPGAs at each process node. Left and right axes show logic cells and memory density in log-scale relative to 250nm. The dotted line shows the cost per logic cell for the highest density FPGA at that node (in cents) where historical pricing was available [84]. The 14-16nm node introduced FinFETs, which greatly increase performance/W, so that the same application may use fewer logic cells. * Data for 7-10nm projected from [22].

1 Introduction

FPGAs offer compelling hardware acceleration in application domains ranging from databases [28, 59, 74], finance [54, 70], neural networks [115, 104], graph processing [36, 85], communication [57, 107, 27], and networking [53, 92, 27]. Over the last few decades, FPGA compute density has grown dramatically, cost per logic cell has dropped precipitously (Figure 1), and higher-level programming abstractions [60, 19, 32, 20, 65, 90] have emerged to improve programmer productivity. Cloud providers such as Amazon [1] are offering compute resources with FPGAs. However, system software has not

kept up. The body of research effort on FPGA OS support [77, 100, 99, 86, 45, 52, 29] and sharing [30] has yielded no first-class commodity OS support, and on-demand FPGAs from AWS and Microsoft support a single-application model.

Current proposals for FPGA sharing [30, 26, 42, 110, 63] partition a physical FPGA into a small number of fixed-size *slots* and demand-share them across user logic using hardware support for *partial reconfiguration* (PR). PR changes the configuration of FPGA fabric within a slot without perturbing the state of the rest of the FPGA. User logic is pre-compiled to a *bitstream* that targets the pre-defined slots, enabling a system to deploy user logic with low latency. A reserved partition of the fabric, or *shell*, implements library support. Fixed-slot designs have significant drawbacks in practice. Forcing applications to target fixed partitions unnecessarily constrains them: the size of the largest partition places an artificial limit on application size, and oversized partitions result in wasted FPGA resources and reduced concurrency.

We present a design and prototype of protected sharing and cross-platform compatibility for FPGAs called AMORPHOS. AMORPHOS enables applications to scale dynamically in response to load and availability, and enables the system to transparently change mappings between user logic and physical fabric to increase utilization. AMORPHOS avoids fixed-size slots and manages physical fabric in dynamically sized **zones**. Zones are demand-shared across *morphable tasks*, or **Morphlets**. A **Morphlet** is a new abstraction which forms a protection boundary and encapsulates user FPGA logic in a way that enables it to be dynamically scaled and remapped to the physical fabric. Morphlets express scalability dimensions and resource constraints using a **mesh**. A mesh is a map from feasible resource combinations to abstract descriptions of the logic. Meshes act as an intermediate representation (IR) that can be re-targeted at runtime to different hardware allocations, allowing the AMORPHOS scheduler to re-target Morphlets to available FPGA fabric. AMORPHOS caches dynamically generated bitstreams in a shared **registry** to hide the latency of re-targeting. AMORPHOS mediates Morphlet access to OS-managed resources through a **hull**, which hardens and extends a traditional shell design with access control and support for isolation. The hull also forms a canonical interface that enables Morphlets to be portable.

We prototype AMORPHOS on both Amazon F1 and Microsoft Catapult. Measurements show that AMORPHOS's abstractions provide both compatibility and protected sharing while dramatically improving utilization and throughput. We make the following contributions:

- A minimal set of OS-level abstractions and interfaces to enable mutually distrustful FPGA sharing and protected access to OS-managed resources.
- A compatibility layer that enables portability of FPGA code across Amazon F1 and Microsoft Catapult FPGA systems.
- Techniques that transparently transition between scheduling modes based on fixed and variable zones to increase utilization and throughput.
- Evaluation of a prototype showing AMORPHOS sharing support increases fabric utilization and system throughput up to 4× (Catapult) and 23× (F1) relative to fixed-slot sharing and non-sharing designs.

2 Background

Field Programmable Gate Arrays (FPGAs) are circuits that can be configured post-manufacture to implement custom logic. FPGAs may be deployed in a system in several ways:

Discrete. A FPGA can be used on its own without a processor. Network switches, for example [17], can be implemented this way to provide a programmable data plane.

System-on-chip. FPGAs may include one or more hard (in-silicon) processors [35, 16] tightly integrated with the FPGA. Logic in the FPGA can manipulate the processor and vice versa (e.g. FPGA logic may directly write into processor caches or manipulate memory controllers).

Bump-in-the-wire. FPGAs can be placed on an I/O pipeline to "transparently" manipulate data. For example, an FPGA may be integrated into a network card or memory and storage controller to provide line-rate encryption [8].

Co-processor/Offload. FPGAs can be I/O-attached (e.g. via PCIe) to offload compute. An application configures the FPGA to implement a hardware accelerator and sends data and requests to it like a co-processor. Many workloads targeting on-demand cloud FPGAs [1, 79], such as DNNs [83, 116], media transcoding [9], genomics [6], real-time risk modeling [87], and blockchain [105, 49] fall in this category. AMORPHOS *is designed for FPGAs deployed in the co-processor/offload configuration.*

2.1 Software versus Hardware

Writing Hardware. Hardware description languages (HDLs), such as Verilog [106] and VHDL [21], enable developers to configure the various resources on the FPGA *fabric*: interconnect, look-up-tables (LUTs), flip-flops, on-chip memory (block RAM), and "hard resources" (adders, DSPs, memory controllers, etc.). Unlike software, where

resource arrangement is abstracted away by the ISA, hardware gives developers explicit control over arranging and connecting resources in a flexible manner.

Building and deploying hardware. To be deployed on an FPGA, a design must be converted into a *bitstream*, a binary which configures the FPGA fabric. The bitstream is built from the HDL in two stages: First, *synthesis* converts and maps the HDL into a *netlist*, which describes how resources on the FPGA should be connected to implement design logic. Synthesis is similar to software compilation and usually takes on the order of minutes. Second, the *place-and-route* (PAR) step takes the netlist and attempts to route the design on the FPGA fabric. PAR is a constraint-solving problem which can take hours for a complex design. A bitstream takes 10s-100s of milliseconds to be loaded.

Sharing and reconfiguring hardware. Unlike software, which can be context switched by saving and restoring architectural state, context switching FPGA hardware at arbitrary points requires capturing the current state of the logic, as loading a new bitstream will reset that state. While mechanisms do exist, they are not universally supported [47] or are in their early stages [23], and are not supported in all AMORPHOS's target environments. Therefore, time-sharing must either be non-preemptive, or must forcibly revoke access to the FPGA, potentially at the cost of losing application state.

Partial Reconfiguration. Hardware support for *partial reconfiguration* [76] (PR) enables parts of an FPGA to be reconfigured *in situ* without impacting the live configuration or circuit state of other parts of the FPGA fabric. Use of the feature necessitates including partial reconfiguration logic along with the netlist during the place-and-route build phase, but does not otherwise impact the process in a fundamental way: the output is a bitstream that targets *a specific set of physical FPGA resources*. Partial reconfiguration can be faster because partial bitstreams are smaller. Because PR can allow an application to change without impacting the state of other applications, it is an attractive primitive for implementing context switching.

Scaling Hardware. Unlike software, which is scaled by increasing the number of cores or the number of operations executed per instruction (SIMD), hardware can scale by implementing what can be thought of as entirely new specialized instructions or algorithms. This enables FPGAs to provide energy-efficiency and evolvability that are difficult to achieve with fixed-function hardware like GPUs or TPUs [117, 46, 11]. For example, a deep neural network (DNN) can be implemented as thousands of independent 2-bit bitwise processors, rather than consuming the pipeline of a general purpose 64-bit processor.

2.2 FPGA OS and Sharing Support

On-demand FPGAs in the cloud, such as Amazon F1 [1], only enable coarse-grain sharing of a FPGA. F1 provides developers with SDKs for developing, simulating, debugging, and compiling hardware accelerators on-demand. FPGA designs are saved as Amazon FPGA Images (AFIs) and deployed to an F1 instance. The AWS Marketplace functions as a library of pre-built common AFIs. At deployment, an AFI is assigned the fabric of the entire FPGA: there is no support for sharing across protection domains. The lack of fine-grained sharing means that both the cloud provider and the user are locked out of the flexibility of the FPGA: once a user loads an AFI, Amazon must assume that the entire FPGA is being used by that AFI, even though the FPGA may be idle. Other than decommissioning the instance, the user has no way to release FPGA resources back to the cloud provider. As a result, workloads which need to conditionally or occasionally offload compute [97], or which cannot fully utilize the FPGA, may be unable to cost-effectively use cloud FPGAs.

Previous proposals have touched on OS-level concerns such as cross-application sharing [31, 109, 52], hardware abstraction layers [111, 61, 78, 62, 50, 80], and shared runtime support [45, 103, 37], or access from a virtual machine [88]. Theoretical aspects of spatial scheduling on FPGAs [43, 102, 108, 31], task scheduling in heterogeneous CPU-FPGA platforms [25, 102, 108, 44, 18], mechanisms for preemption [73], relocation [55], and context switch [72, 93] are well-explored. Access from an FPGA to OS-managed resources such as virtual memory [33, 15, 114, 77], file systems [100], and system calls [77, 100] has enjoyed the research community's attention as well. However, no first class OS support for FPGAs is present in modern commodity OSes and cloud FPGA platforms support a single application model.

Recent designs for FPGA sharing in datacenters [30, 26, 42, 110, 63] leverage partial reconfiguration to demand share fixed pre-reserved partitions of FPGA fabric among applications with bitstreams pre-compiled to target those partitions. AMORPHOS begins with a design of this form, extends it to enable cross-domain protection, and replaces the fixed slot restriction with elastic resource management to increase utilization and throughput.

3 Goals

AMORPHOS supports demand-sharing of FPGAs by mutually distrustful processes. AMORPHOS multiplexes fabric spatially by default, co-scheduling user logic from different processes, and falling back to time-sharing when

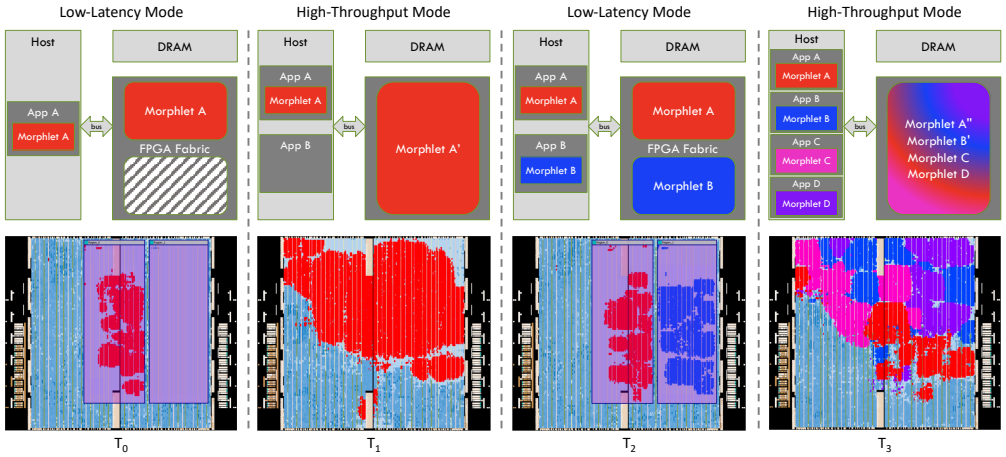


Figure 2: AMORPHOS managing a number of DNNWeaver (see §6) Morphlets. The top row depicts the host and FPGA state while the bottom row shows the corresponding chip layout on Catapult. At T_0, a single DNNWeaver Morphlet is placed on the FPGA. At T_1, AMORPHOS detects underutilization and transitions to high-throughput mode, giving the Morphlet more area. At T_2, another Morphlet is instantiated and AMORPHOS returns to low-latency mode. Finally, at T_3, 2 more DNNWeavers have been scheduled and AMORPHOS transitions to high-throughput mode to fit them all on the FPGA.

space-sharing is infeasible due to resource constraints. A critical design objective for AMORPHOS is avoiding the artificial constraints on inter- and intra-Morphlet scalability induced by a fixed-slot design. AMORPHOS enables individual applications to utilize additional fabric if available, and enables multiple applications to share the fabric to achieve higher aggregate utilization.

3.1 Programming Model

We target a programming model of HDL (hardware description language) over an abstract FPGA fabric. The primary tangible change from current HDL-FPGA programming models is the requirement for the developer to use virtual interfaces for communication with the host and access to on-board resources such as DRAM, network I/O, etc. Collectively, these interfaces form a mediation and compatibility layer called the hull, which encapsulates, hardens, and extends current vendor-specific *shells* [92, 1]

3.2 Isolation

AMORPHOS provides protection guarantees similar to those provided to processes in a software OS. Memory and I/O protection is enforced between Morphlets. Best effort performance isolation is provided based on resource allocation policy and scheduler hints. When FPGA resources are constrained, AMORPHOS dedicates an even share of I/O and memory bandwidth to each Morphlet, enforced by a hardware arbiter. AMORPHOS makes a best effort to allocate fabric fairly under contention by preferring spatial assignments that balance the resources allocated to each application, and time-slicing fairly when spatial sharing is unfeasible (see §4.2 for details). Extending these mechanisms to provide priority-proportional fairness is straightforward, but our prototype currently does not provide flexible software-exposed policies, which we leave as future work. Our current design avoids co-scheduling Morphlets which will interfere with each other through contention on the hardware based on scheduler hints.

AMORPHOS does not provide explicit protection against side channels. Side channels exist and are an active area of research where some mitigations now exist [94]. However, the attack surface for Morphlets is considerably smaller, as Morphlets enjoy exclusive access to all the FPGA hardware resources they use except interfaces to AMORPHOS itself, which are implemented with cross-domain isolation in mind. For example, special care is taken to zero out all signals on a Morphlet's interface if it is not the intended recipient of a transaction, which ensures the Morphlet can not monitor the address/data signals of other Morphlets.

3.3 Dynamic Scalability

A key goal of AMORPHOS is increasing utilization. When only a single application is on the FPGA, it should enjoy exclusive access to all resources it can actually use. When multiple Morphlets contend, if a feasible partitioning of the fabric accommodating them all exists, applications are mapped to shares of the fabric concurrently. If no feasible partitioning exists, the system falls back to time-sharing at coarse granularity. A key challenge to realizing this vision is very high latency (potentially hours or more) of place-and-route (PAR), which maps user-logic to physical fabric. Using partial reconfiguration (PR) to deploy appli-

cations avoids that latency, but constrains applications to fixed slots, giving up elasticity. Avoiding or hiding PAR latency without constraining logic to fixed slots is a primary design goal for Morphlets and the AMORPHOS scheduler. Furthermore, for Morphlets to take advantage of different size partitions, the programming model must provide a way for the developer to express scalability dimensions, valid configurations, and hints to the system to inform the scheduler.

While AMORPHOS's primary sharing strategy is spatial sharing, support for time-sharing is a *de facto* requirement to avoid starvation when the FPGA is contended. Preemptive time-slicing requires mechanisms for capturing, evacuating, and restoring state on the FPGA, and while some applicable mechanisms do exist (e.g. ICAP [47]) they are not universally supported, and state-capture remains an active research area [55, 72, 93, 73]. We opt for a non-preemptive context switch based on extensions to the programming that include a *quiescence interface*.

3.4 Motivating Example

Figure 2 shows a series of scheduling decisions taken by our system in response to applications requesting use of the FPGA. The top row depicts the state of the host and FPGA while the bottom row shows the corresponding chip layout on Catapult V1 FPGAs [92] (Altera Stratix V 5SGSMD5H2F35I3L). At time T_0, process A instantiates a Morphlet on the FPGA. To provide on-demand access at the lowest latency, it initially deploys A on fixed-size zone 1 using partial reconfiguration. At time T_1, AMORPHOS notices the resulting under-utilization and *morphs*. A's mesh is used to select a more performant netlist that uses as much of the FPGA as it can profitably consume, and full reconfiguration is used to give A all the resources not consumed by AMORPHOS itself. At time T_2, process B requests FPGA fabric. To serve that demand quickly, AMORPHOS *morphs* again, reinstating A in zone 1, and mapping B to zone 2. At some future time T_3, which represents the state after potentially many intervening events, four processes have requested FPGA access, and AMORPHOS has *morphed* by selecting netlists from each Morphlet's mesh to produce a single combined bitstream that co-schedules all. Utilization and throughput are improved by 2× compared to a fixed slot design.

4 Design

AMORPHOS introduces a number of new abstractions and interconnected components. A system overview is shown in Figure 3. User logic is encapsulated in **Morphlets**, a **zone** manager tracks allocatable area of physical FPGA fabric, and a scheduler manages the mapping between Morphlets and zones. To enable flexible mapping of Morphlets to zones, Morphlets encapsulate information to enable the scheduler to generate new bitstreams on demand, in the form of **meshes**. To hide the latency of PAR for dynamic re-targeting of Morphlets, the scheduler maintains a **registry** that caches (potentially combined) bitstreams that can be instantiated on a zone with low latency. AMORPHOS mediates Morphlet access to memory and I/O with a compatibility and protection layer called the **hull**.

4.1 Hull

The primary job of the **hull** is to provide cross-domain protection by mediating access to memory and I/O, and to enable compatibility by presenting Morphlets with canonical interfaces to interact with the rest of the platform. The hull coordinates with the scheduler by sending and monitoring quiescence signals (§4.3), disabling connections to zones of the FPGA currently being reprogrammed (§4.2), and connecting and initializing Morphlets after reprogramming is complete. The hull provides memory protection for on-board DRAM using segment-based address translation and manages peripheral I/O devices by implementing shared logic to interface with them, along with simple access mediation logic (e.g. rate-limiting for contended I/O). Finally, the hull exports interfaces to the host OS to configure access control and protection mechanisms, e.g. base and bounds registers for segments.

We expect that future FPGA platforms will provide some of this functionality, address translation in particular, in "hard IP," meaning it will be supported directly in silicon. Our current prototypes are forced to synthesize these functions from FPGA fabric.

4.2 Zones and Scheduling

The **zone** manager allocates physical FPGA fabric to Morphlets. Fabric not consumed by the hull forms a *global* zone, which can be recursively subdivided into smaller reconfigurable zones that can be allocated to different Morphlets. Our Catapult prototype supports two smaller zones within the global zone, each of which can be further subdivided into two. F1 hardware has considerably more resources, and could support a considerably larger number of zones with more levels of subdivision. However, F1 does not expose the partial reconfiguration feature, so our F1 prototype is forced to manage only a single global zone. Zones may be allocated to individual Morphlets or may accommodate multiple Morphlets simultaneously. When it is time to schedule a Morphlet, the job of the zone manager is to find (or create) a free reconfigurable zone matching the Morphlet's default bitstream. If a match is found, the Morphlet can be deployed on that zone di-

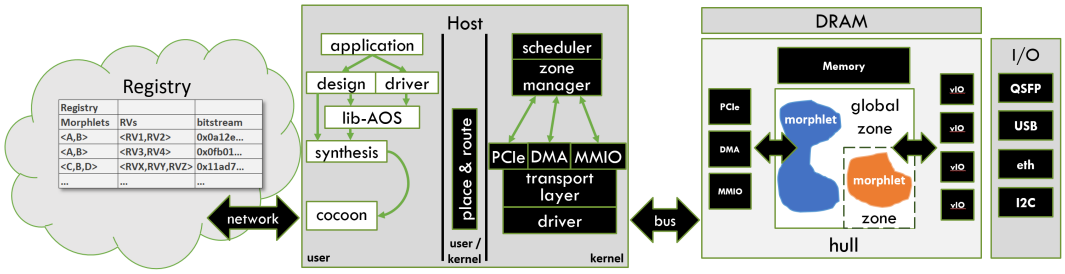


Figure 3: AMORPHOS design overview. FPGA Morphlets (applications) are synthesized by the user and given to AMORPHOS to be converted into bitstreams capable of being placed on the FPGA. The FPGA is split into a hull and multiple zones, in which Morphlets can be scheduled from cocoons. Access to memory and I/O from Morphlets is virtualized by the hull, which implements the logic to interface with the resources directly and to ensure proper access control. On the host side, communication to the Morphlet is virtualized through the lib-AMORPHOS interface.

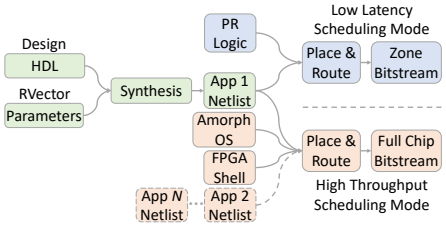


Figure 4: AMORPHOS Morphlet life cycle.

RVector	Cocoon

```
struct {
  bool optLatency;
  size_t minMem;
  size_t optMem;
  size_t memBw;
  size_t PCIeBw;
} RVector;
```

```
struct {
  bitstream default_bitstream;
  map<RVector, netlist> mesh;
} Cocoon;
```

Figure 5: Object model for Cocoons.

rectly. If one is not found, the zone manager must coalesce free (or reclaimed) zones to form a larger one, and inform AMORPHOS that it must re-target the Morphlet along with any other currently-running Morphlets to be deployed on the coalesced zone. In the limit, all Morphlets are deployed together on the global zone, maximizing aggregate utilization and individual application performance.

Zones play a key role in balancing scheduling latency against aggregate throughput because fixed zones and PR is better for fast deployment, while a larger zones with multiple Morphlets is better for utilization and throughput. AMORPHOS's scheduler supports two modes reflecting this tradeoff, *low-latency mode* and *high-throughput mode*, and transitions between those modes transparently based on demand.

In low-latency mode, reconfigurable zones enable Morphlet to be deployed almost instantly through partial reconfiguration with the Morphlet's default bitstream. The Morphlet's default bitstream targets one or more of the smaller zones and includes the partial reconfiguration logic required to enable it to use PR. PR-based scheduling also allows other Morphlets to continue uninterrupted. However, reconfigurable zones incur significant area overhead for the additional PR logic required and increase fragmentation of the FPGA fabric.

When the reconfigurable regions cannot accommodate the Morphlets of all applications concurrently, a *morph* operation occurs. The zone manager coalesces zones to form larger ones, eventually converging to the single global zone, and the scheduler enters high-throughput mode. To do so, it re-targets running Morphlets by running place-and-route to create a bitstream that includes logic for all of them and subsequently maps that bitstream to the target zone. When the global zone is the target, this requires reconfiguring the whole FPGA. However, the global zone can accommodate significantly more Morphlets because PR support fabric is freed, and fragmentation is eliminated by not restricting Morphlets to exclusive partitions of the FPGA. AMORPHOS hides the latency of place-and-route for *morph* operations by caching or pre-computing combined bitstreams targeting the global zone in a Morphlet registry. The registry's entries are bitstreams for "co-Morphlets" representing co-compiled combinations of Morphlets.

AMORPHOS also uses the *morph* operation for single Morphlets when the FPGA fabric is underutilized. Moving a Morphlet to a larger zone or the global zone puts significantly more resources at its disposal. The application can then use these resources to run faster. AMORPHOS targets applications in which Morphlets will likely run for an extended time, so the overhead of moving Morphlets to larger zones is amortized by the gains in aggregate throughput. The ability of a Morphlet to benefit from increasing resource shares is visible to AMORPHOS through the Morphlet's mesh, enabling AMORPHOS to avoid *morphing* when it is not performance profitable to do so.

4.3 Morphlets and Cocoons

While **Morphlets** are analogous to and extend the process abstraction, the AMORPHOS build toolchain produces **Cocoons** from HDL specifications targeting AMORPHOS, which are analogous to an application binary. In addition to the deployable bitstream produced by current FPGA build tools, Cocoons encapsulate abstract information about the Morphlet to enable stages of the build toolchain to be re-invoked dynamically to produce different bitstreams on demand. Dynamic re-targeting enables co-scheduling of multiple Morphlets on a zone or dynamic scaling of the fabric resources allocated to an individual Morphlet.

Figure 5 shows the contents of a Cocoon, and Figure 4 shows how the various stages in the build and deployment process interact with Cocoons to enable dynamic targeting. A cocoon's *default bitstream* targets a default zone on the device and can be deployed using PR. Its **mesh** encapsulates a constrained set of strategies for re-targeting the Morphlet's user logic. Concretely, a mesh is a map of abstract descriptions of the logic, or **netlists**, keyed by **RVectors**. An RVector describes a feasible combination of resources and scheduler hints for the corresponding netlist. The netlist acts as an intermediate representation (IR) which can, potentially in combination with netlists from other Morphlets, be used as input to place-and-route tools to produce new deployable bitstreams. The *default bitstream* is always used when the scheduler is in low-latency mode. When the scheduler is in high-throughput mode it may compare current system state against RVectors in the mesh to select an appropriate netlist. To deploy the dynamically chosen configuration, the scheduler can then produce the required bitstream or look it up in in the Morphlet registry (§4.5) to hide place-and-route latency.

RVectors. A RVector (Resource Vector) describes Morphlet resource constraints and utilization hints that cannot be derived from the netlist in the mesh. Important entries include Boolean valued hints for memory and PCIe usage which simplify connection to AMORPHOS FPGA-side interfaces, as well as optimal and minimal memory footprint and bandwidth estimates. Our experience implementing AMORPHOS is that hints regarding an application's bottleneck resources and access patterns are essential to guide co-scheduling. For example, this allows the hull to be optimized for lower memory access latency with some bandwidth trade-off. Note that low level FPGA-specific resources (e.g. number of LUTs, BRAMs, etc.) can be derived from a netlist and are not included in a RVector.

Quiescence Interface. Evacuating Morphlets from the FPGA is necessary when the enclosing process terminates

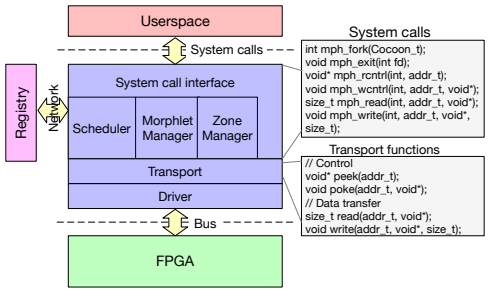


Figure 6: AMORPHOS host stack interfaces between user space and FPGA Morphlets.

or when the scheduler needs to reallocate a zone to another Morphlet. Rather than immediately removing the Morphlet (at risk of losing work) or attempting to capture and save a Morphlet's state (difficult with current hardware [98, 56]), the hull provides a quiescence interface to inform the Morphlet of the impending context switch. The Morphlet is then given an opportunity to enter a stable state and/or save its progress. A Morphlet informs AMORPHOS that it can be safety switched by asserting a *quiescence* signal through the hull. Unresponsive Morphlets are forcibly evacuated after a configurable time-out to avoid DoS. Our current design allows Morphlets to leave data in on-board memory in the absence of memory pressure from incoming Morphlets. Transparent swap in/out of a Morphlet's FPGA DRAM state is a straightforward operation; our current prototypes do not yet support it.

4.4 Host Stack/OS interface

AMORPHOS integrates with the OS in the Catapult stack and acts as a user-mode library for F1. The entire host stack is depicted in Figure 6. AMORPHOS's OS interface exposes system calls to manage Morphlets and enables communication between host processes and Morphlets. The interface provides APIs to load and evacuate Morphlets as well as to read and write data over the transport layer to FPGA-resident Morphlets.

4.5 Morphlet Registry

AMORPHOS dynamically transitions between low-latency and high-throughput scheduling mode, reflecting a fundamental latency/density tradeoff. To hide the latency of dynamic bitstream generation, AMORPHOS maintains a registry, a cache of precomputed bitstreams that contain deployable spatial sharing combinations of multiple Morphlets. For a large number of Morphlets, precomputing bitstreams for all possible combinations is impractical, particularly when combinations include duplicate Morphlets. We argue that a number of factors enable us to reduce the search space to a practical level. First, building

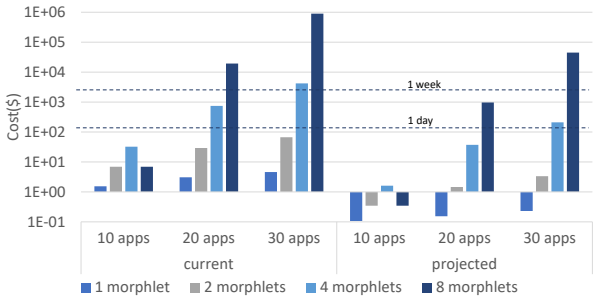


Figure 7: Cost of pre-compiling all possible combinations of Morphlets given varying numbers of deployable Morphlets and varying levels of concurrency. Cost is in dollars and reflects the cost of renting demand infrastructure from Amazon AWS to run the build toolchains. "Current" data are based on measurements with our present toolchain, while projected are scaled to assume a (conservative) 20× improvement in place-and-route performance based on [40, 39, 38].

combined Morphlets can occur in parallel. Second, reducing the latency of place-and-route is an active area, and recent research has produced order of magnitude reductions (20-70×), e.g based on GPUs [40] or other parallel resources [39, 38]. Third, Morphlets can be grouped by popularity or according to hints encoded in RVectors to bound the the number of choices, and sharing densities need not be maximized to achieve multiplicative improvements in throughput and utilization.

Figure 7 shows the cost in dollars for AWS infrastructure to pre-compile all possible combinations of Morphlets for varying numbers of Morphlets and concurrency levels using current tools and using future tools whose performance is projected based on [40, 38, 39]. Compile times are derived from our own benchmark builds. [1] The dotted lines correspond to a day and a week of compute time on 20 VMs. The AWS marketplace, at the time of this writing, offers only 18 FPGA applications [4]. From this pool, all possible co-schedules of 4 Morphlets can be computed in under a day for $100 in computation time. Faster future build tools and careful grouping to reduce the search space can increase utilization further. For example, if co-locatable Morphlets are partitioned in groups of 20, all densities of up to 8 can be precomputed in a handful of days for $1,000. The registry need not eliminate lookups or maximize density to significantly improve utilization.

5 Implementation

We implement AMORPHOS on Amazon F1 FPGA cloud instances[1, 2] and the Microsoft Catapult open research platform [92], available at TACC [5].

[1] Concretely, a single instance of DNNWeaver can be compiled for F1 in 103 minutes. The second and third instances bring that to 118 minutes, while 8 instances can be co-compiled in 157.

Catapult and F1 both support shells to provide three basic forms of platform library support: 1) a bulk host-FPGA data transfer interface, 2) a control interface to manage FPGA applications, and 3) interfaces to on-board DRAM. Catapult and F1 expose these functions with different levels of abstraction. Catapult supports packetized bulk data transfers, a register interface for control signals, and a simple FPGA-side memory read/write interface with independent ports. F1's shell exports AXI4 [3] interfaces to encapsulate these three functional areas. AMORPHOS's interface must encapsulate both Catapult and F1 interfaces, as well as implement address translation for memory protection and I/O access mediation.

The AMORPHOS hull exposes 1) Control Register (CntrlReg) for Morphlet management, 2) Simple PCIe for bulk data transfer, and 3) a AMORPHOS Memory Interface (AMI) supporting 64-byte read/write transactions. Morphlets written to these interfaces are portable across Catapult and F1. AMORPHOS transparently manipulates address bits so Morphlets believe they have full control of memory. OS-programmable BARs (base-address registers) are used to control and protect what regions of memory are accessible to different Morphlets. In addition to memory protection, AMORPHOS provides each Morphlet with a virtual address space and abstracts away the 1-to-1 port-to-channel mappings imposed by F1 and Catapult shells. Virtual address spaces are striped across all memory channels. The number of co-resident Morphlets, memory access ports per Morphlet, and number of memory channels are parameters for the hull. Furthermore, the hull is modular and incurs no overhead for unused interfaces on a target FPGA platform.

Logic structures, such as FIFOs, are fundamental building blocks for FPGA application designers. AMORPHOS provides an FPGA-agnostic wrapper, HullFIFO, that exposes a high level interface to efficiently map to low-level primitives on both F1 and Catapult.

5.1 Catapult

Catapult divides FPGA fabric into a *shell* and user-logic called a *role*. The Catapult shell interface to memory is two 64-byte wide read/write ports over *disjoint* address spaces. AMORPHOS adopts the 64-byte transaction size but virtualizes the interfaces for multiple co-resident Morphlets using segment-based address translation and buffering to support application-level read-modify-write operations.

To enable AMORPHOS to use partial reconfiguration to manage zones we add a PR controller and PR wrapper. The PR controller streams in PR bitstream data from the PCIe bus and transfers it to a PR IP module (vendor-provided Intellectual Property logic block) which uses it

to reconfigure the zone fabric. I/O to each Morphlet is routed through the PR wrapper, which handles driving the Morphlet inputs and disconnecting the Morphlet outputs during PR. This safeguards the application and prevents spurious I/O during the programming process.

5.2 F1

F1 features a *shell* and a user application as *Custom Logic (CL)*. F1 features twice as many memory channels as Catapult and requires the CL to instantiate additional memory controllers if more than one memory channel is needed. AMORPHOS handles instantiating the memory controllers and is parameterized to scale itself to handle additional memory channels. The F1 shell features many different PCIe interfaces, some for DMA type transfers between the host and some lower throughput for management/-control of the CL. PCIe and Memory on F1 are exposed over AXI4 interfaces, which are more complex than the interfaces on Catapult. This complexity is abstracted away from the Morphlet and implemented in our hull. The hull sits on top of an unmodified F1 shell.

5.3 Multiplexing AMORPHOS Interfaces

Large numbers of concurrent Morphlets can stress AMORPHOS's internal FPGA-side subsystems. Each Morphlet requires the same set of interfaces (CntrlReg, Memory, and PCIe). Routing and connections to all of them is complicated by the fact that I/O pads for each can be (and are on F1/Catapult hardware) on different edges of the physical FPGA, which stresses place and route tools by complicating the routing problem and increasing congestion. Designing AMORPHOS's multiplexing logic to anticipate scale can mitigate some, but not all, of the problem. An initial design used multiple flat multiplexers to distribute interface signals to each Morphlet, but we found that, despite plenty of available fabric, they could not scale past 4 concurrent Morphlets in most cases.

Our current design implements a pipelined binary tree to route the CntrlReg signals. The tree-distribution network enables us to add pipeline stages, making it easier to meet timing while reducing the fanout of large data buses. The benefit is a substantial improvement to the scale at which AMORPHOS can route interfaces to concurrent Morphlets. The trade-off is minimal additional latency: 1 additional cycle for each layer, easily tolerable for CntrlReg, which is a low-bandwidth control interface.

Our current implementation takes a different approach with memory. Rather than scale the memory subsystem to provide N Morphlets with access to M memory channels for an arbitrary number of Morphlets, AMORPHOS uses flat multiplexing with up to 8 Morphlets and statically partitions the memory channels across *groups* of Morphlets at sharing densities above 8. This policy enables us to use a single-level of multiplexing and provide access to all channels for all Morphlets at lower densities but avoids the complexity and latency of an additional tree network at high densities. The tradeoff is that Morphlets are restricted to using a subset of DRAM channels, which does not alter the capacity of their memory share but does reduce the bandwidth available to them. Memory systems perform better when they manage fewer access streams (assuming sequential access) because back-to-back operations from a single stream enable optimizations that are not feasible between operations from different streams. The design decision enables much higher densities as it improves routability: a group of Morphlets only need to route to a subset of the memory channels. Our experience is that memory bandwidth contention determines the upper bound on scalability for Morphlets which share DRAM. Contention occurs at lower levels of concurrency than the levels that require strict group-based DRAM channel partitioning, so optimizing DRAM access for high sharing density is unlikely to provide substantial benefits.

5.4 Host Stack

AMORPHOS provides a host stack which interfaces with userspace applications, implemented as an OS extension in our Catapult prototype, and as a user-mode library for F1. The host stack comprises a system call interface, FPGA Morphlet manager and scheduler, zone manager, and transport layer that encapsulates the control and bulk transfer interfaces described above (§5). The interface and stack structure are illustrated in Figure 6. Control signals and reading/writing data are passed through the syscall interface to the transport layer. Morphlet allocation, scheduling hints, and tear down are redirected to the Morphlet scheduler and zone manager.

The host system call interface for Catapult is implemented as a service which supports the transport layer by wrapping the Catapult driver and library stack. The service associates Morphlets with file descriptors, exporting read and write operations on them, and communicates with the scheduler to monitor the active state of executing Morphlets or request quiescence.

6 Evaluation

AMORPHOS runs on both a Mt Granite FPGA board in the Catapult V1 cloud platform [92], containing an Altera Stratix V GS running at 125 MHz with two 4 GB DDR3 channels, and an Amazon F1 cloud instance [1], using a Xilinx UltraScale+ VU9P running at 125 MHz with four 16 GB DDR4 channels. Both platforms are connected over a PCIe bus and support build tools we adapt to build AMORPHOS and our benchmarks, summarized in Table 1.

Program	Description
DNNWeaver	Convolutional neural network
MemDrive	Memory streaming
Bitcoin	Bitcoin hashing accelerator
DFADD	Double-precision addition
DFMUL	Double-precision multiplication
DFSIN	Double-precision Sine function
MIPS	Simplified MIPS processor
ADPCM	Adaptive differential pulse codec
GSM	Linear predictive coding analysis
JPEG	JPEG image decompression
MOTION	Motion vector decoding
AES	Advanced encryption standard
BLOWFISH	Data encryption standard
SHA	Secure hash algorithm

Table 1: Benchmarks used to evaluate AMORPHOS

Catapult Benchmark	Logic Cells	Registers	BRAM Bits
DNNWeaver	39,994	108,640	387,840
MemDrive	2,449	1,488	570,496
Bitcoin	42,171	60,257	21,408
blowfish	20,581	24,082	810,850
gsm	20,910	24,716	5,552
mips	17,672	19,981	657,574
dfmul	17,759	20,586	0
aes	23,900	28,366	689,630
motion	25,178	26,734	687,366
dfadd	18,043	21,014	662,694
sha	17,772	21,380	788,806
adpcm	22,840	29,837	663,654
jpeg	42,243	40,327	1,116,312
dfsin	26,742	32,572	663,805
F1 Benchmark	**LUTs**	**Flip Flops**	**BRAM Bits**
DNNWeaver	4,924	4,773	339,968
MemDrive	1,136	930	0
Bitcoin	40,106	46,191	0

Table 2: FPGA resource utilization by Morphlet type broken down by resource type as reported by each platform.

Benchmarks. We evaluate benchmarks that cover three important categories for FPGA applications, defined by whether they are *memory-bound*, *compute-bound*, or *dynamic resource bound*. Morphlets are *compute-bound* when low-level FPGA resources such as LUTs, BRAMs, etc. are limited. Morphlets are *memory-bound* when off-chip memory bandwidth or latency constrains their performance. Morphlets are *dynamic resource bound* when they can be mapped to the fabric in ways that represent different points along their roofline model [82], meaning they can be *memory-* or *compute-bound*. Our Bitcoin Morphlet (based on [12]) is *compute-bound*. It is parameterized to replicate hashing units and can scale to consume most of the on-board FPGA fabric. Additional instances of functional units increase logic utilization limiting the maximum size/throughput of the Morphlet. Applications that are *memory-bound* usually have a low compute-to-memory ratio and directly benefit from additional off-chip memory bandwidth. Streaming applications (e.g. in database [75] or search [112]) access large amounts of data, often discarding much of it or doing minimal compute per datum. To represent a range of such applications, we wrote a custom Morphlet called MemDrive (MemD) that can be configured on the host side post-synthesis to generate different memory traffic patterns and read/write ratios, along with operations such as fills, reductions, and ECC checks.

Many applications can be configured to take advantage of either additional logic or additional memory bandwidth, corresponding to different points along their roofline model. To represent this class, we evaluate DNNWeaver [96], an open source DNN design framework that can be used to synthesize models from a description of a specific network topology. The user controls the number of functional units and data buffer sizes, translating to variable demand for on- and off-chip resources. We instantiate DNNWeaver with an 8-layer LeNet [71] topology.

To increase benchmark diversity, we include a number of benchmarks that perform many useful non-trivial functions that do not fully utilize the fabric or memory bandwidth. We use the LegUp [7] high level synthesis (HLS) environment to generate 11 Morphlets (a subset of CHStone[48]). LegUp applications use memory by composing it from BRAMs when needed, rather than off-chip DRAM, so they do not contend for DRAM bandwidth. However, as many FPGA applications (DNNs included) are optimized to ensure their working set fits in on-chip BRAMs to minimize off-chip memory access, we believe they are representative.

Metrics. We report resource utilization and performance measured by throughput. The build tools for each platform break down resource utilization into logic, registers/flip-flops, and BlockRAMs. Morphlets are instrumented with cycle counters to measure the runtime on the FPGA when each is running. End-to-end execution time is measured from the host side. Performance for MemDrive is reported as memory throughput (bytes/cycle). Bitcoin performance is reported as normalized hash throughput with the baseline being a fully unrolled and pipelined instance of the application (to the maximum the open source code permitted), producing a full block hash per cycle. DNNWeaver performance is reported as normalized throughput, where the baseline is the number of cycles required for input data to run through all network layers and complete inference.

We evaluate AMORPHOS with 14 different benchmarks, listed in Table 1. The logic, register, and memory utilization of these benchmarks is listed for both Catapult and (partially for) F1 in Table 2.

Table 3 shows increases in utilization and system

Catapult Configuration	# ALMs	Utilization	Sys. Throughput
1 Bitcoin	63,973	1.00x	1.00x
2 Bitcoin	93,908	1.47x	2.00x
4 Bitcoin	141,139	2.21x	4.00x
1 DNNWeaver	92,619	1.45x	1.00x
2 DNNWeaver	134,972	2.11x	2.00x
4 DNNWeaver	154,956	2.42x	3.31x
1 DNN, 1 MemD	92,135	1.44x	1.41x
2 DNN, 2 MemD	148,249	2.32x	0.80x
1 DNN, 1 BTC	112,010	1.75x	2.00x
2 DNN, 2 BTC	140,635	2.20x	3.68x
1 DNN, 1 BTC, 2 MemD	96,994	1.52x	1.86x
2 BTC, 2 MemD	95,936	1.50x	2.77x
F1 Configuration	**# LUTs**	**Utilization**	**Sys. Throughput**
1 MemD	68,885	1.00x	1.00x
2 MemD	89,161	1.29x	1.67x
4 MemD	100,773	1.46x	1.37x
8 MemD	127,530	1.85x	0.78x
1 Bitcoin	104,851	1.52x	1.00x
4 Bitcoin	229,482	3.33x	4.00x
8 Bitcoin	484,879	7.03x	8.00x
1 DNNWeaver	90,118	1.31x	1.00x
4 DNNWeaver	129,925	1.89x	3.94x
8 DNNWeaver	187,839	2.73x	7.80x
16 DNNWeaver	294,290	4.28x	14.80x
32 DNNWeaver	397,580	5.78x	23.22x

Table 3: Morphlet configurations run in AMORPHOS with corresponding ALM/LUT (logic) usage, relative system utilization improvement, and relative system throughput.

throughput that are made possible by co-scheduling Morphlets using AMORPHOS. Utilization is measured as ALM (adaptive logic module) or LUT (lookup table) usage relative to the smallest configuration on each platform, 1 Bitcoin for Catapult and 1 MemDrive for F1. System throughput is reported as the sum of each Morphlet's normalized throughput, relative to a single instance of that Morphlet. In only two cases does co-scheduling Morphlets result in reduced system throughput, both of which involve multiple MemDrive Morphlets, which interfere significantly with other memory-dependant Morphlets. In the best cases, co-scheduling Morphlets results in 7.03x increased utilization and 23.22x increased throughput.

6.1 CHStone

We evaluate CHStone benchmarks to illustrate generality and to demonstrate that useful accelerators can be co-scheduled at high density with AMORPHOS to increase throughput. We find that the upper bound on density for all is determined by AMORPHOS's ability to route control interfaces to them, which translates to an upper bound of 8 on our Catapult prototype. Because the LegUp compiler implements memory with BRAM, rather than by connecting to on-board DRAM, the CHStone workloads only shared resource is the CntrlReg interface. Absent any source of contention, they scale linearly to the upper bound when co-scheduled as Morphlets on AMORPHOS.

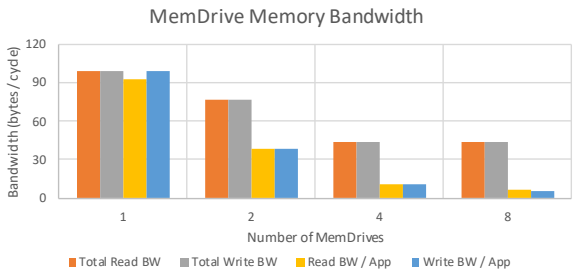


Figure 8: Total and per-MemDrive memory bandwidth for different numbers of Morphlets running in AMORPHOS on Catapult.

We do not report measurements on our F1 prototype as they illustrate the same phenomenon.

6.2 MemDrive

We study contention between *memory-bound* Morphlets using MemDrive, which stresses memory bandwidth. AMORPHOS's 64-byte read/write interface maps well to Catapult, but does not support burst transactions (one transaction returning multiple data payloads), which is necessary to achieve high read throughput on F1's AXI interface to memory. While we were able to achieve peak write-bandwidth on F1 and observe contention due to multiple applications running concurrently, we were unable to saturate read-bandwidth. In future work, our intention is to introduce burst detection and dynamically coalesce memory requests.

Catapult's memory system has a theoretical bandwidth of of 128 bytes/cycle. Experiments on our Catapult prototype show that the total achievable memory bandwidth is roughly 100 bytes/cycle for writes and 90 bytes/cycle for reads. We ran MemDrive in AMORPHOS and directly on the Catapult system to confirm that our virtualization layer incurs no bandwidth loss. Figure 8 shows the per-Morphlet and total system read/write bandwidth when running 1–8 Morphlets of MemDrive in AMORPHOS. Total system bandwidth decreases as the number of co-resident Morphlets rises from 1 up to 4, and saturates from 4 to 8. On F1, we observed similar contention when scaling from 1 to 8 MemDrive Morphlets. The RVector of each Morphlet provides hints to AMORPHOS's on-FPGA memory scheduler, enabling it to manage contention fairly, and improve effective memory bandwidth (e.g. by batching memory requests) or minimize latency for Morphlets that are latency sensitive. MemDrive is not latency sensitive, but its ability to saturate memory has implications for the memory scheduler, which must take care to ensure that latency sensitive Morphlets such as DNNWeaver are not impacted by that saturation.

6.3 DNNWeaver

Table 3 shows how DNNWeaver scales when instantiating multiple Morphlets. We see that aggregate throughput increases with more Morphlets, but contention causes the deviation from perfect linear-scaling to increase with increasing co-resident Morphlets. Both Catapult and F1 theoretically have enough bandwidth to support up to 4 and 32 DNNWeaver Morphlets, respectively. Contention for the memory system manifests as an increase in memory latency for DNNWeaver. We further show this contention in Table 3 by pairing DNNWeaver with MemDrive. Since DNNWeaver performance can suffer if it is paired with a *memory-bound* Morphlet, encoding a Morphlet's sensitivity to memory bandwidth/latency in the RVector is useful for the AMORPHOS scheduler.

6.4 Bitcoin

Up to 4 and 8 Bitcoin Morphlets can be co-resident on Catapult and F1 respectively. Table 3 shows that scaling is linear as Bitcoin only contends for on-chip resources, which are assigned during bitstream generation. The RVector for a Bitcoin-type application specifies that there is no runtime overhead except fabric resources. This would enable AMORPHOS to intelligently co-schedule Bitcoin with other Morphlets that make heavy use of memory but require much less fabric resources, such as MemDrive. *Compute-bound* Morphlets would be great for utilizing unused fabric as they can scale with available logic resources without hurting the performance of *memory-bound* Morphlets. We show this in Table 3 by pairing Bitcoin with DNNWeaver and MemDrive.

6.5 Density Limits

To determine the limits on sharing density we co-schedule as many concurrent Morphlets as possible, manually manipulating the build process where necessary to achieve higher density. While AMORPHOS can achieve high levels of concurrency this way, practically attainable and performance profitable levels are lower. High density co-scheduling of Morphlets stresses build tools because interfaces must be routed to each Morphlet. Avoiding routing congestion at higher densities require manipulation of the build tools. For example, configuring the build tools to focus on congestion rather than logic minimization spreads out the design and replicates logic, increasing area overheads. Routing is heuristic, so successfully meeting timing can depend on trying multiple random seeds. Such interventions are impractical to automate in an OS scheduler, and a production deployment of AMORPHOS would necessarily tolerate sharing densities below the maximum possible.

Morphlet	MaxPerf	MaxTools	Max
DNNWeaver	32	8	32
Bitcoin	8	4	8
MemDrive	2	8	32

Table 4: Limits on AMORPHOS F1 sharing density for DNNWeaver, MemDrive, and Bitcoin. The MaxPerf column indicates the level of Morphlet concurrency at which throughput is maximal. The MaxTools column indicates the maximum concurrency achievable without manual intervention in the build process. The Max column indicates the maximum level we attained with manual intervention in the build process. For example, DNNWeaver's maximal performance is achieved at 32 Morphlets, which is only achievable with manual effort; the build tool chain defaults achieve a maximum density of 8.

Limits on sharing density differ across workloads. Table 4 shows maximum densities on F1 when the upper bound is determined by best throughput, build transparency, or physical limits of the FPGA.

6.6 End-to-End Performance

To compare AMORPHOS against other FPGA sharing designs, we measure the time required to run 1-8 Bitcoin instances on Catapult using AMORPHOS in high-throughput mode, several slot-based approaches, and a no-sharing baseline. The performance of slot-based approaches is emulated by running AMORPHOS in low-latency mode, which uses PR to switch between zones of equal size. The performance of not sharing is emulated by running AMORPHOS with a single Morphlet. Since programming the whole FPGA using the Catapult tools takes a significant portion of time, we also emulate optimal full FPGA reconfiguration by adding a delay of 200ms, which is comparable to programming the whole FPGA via PR. The overhead of using AMORPHOS to emulate these approaches is negligible compared to application runtime, so we expect our results to be accurate for all approaches.

In high-throughput mode, AMORPHOS can fit 4 full-sized Bitcoin Morphlets on the FPGA: we assume that the registry is pre-populated with the required bitstream (see §4.5). When using fixed slots, only two Bitcoin instances can be co-resident. Since slots may not always be able to fit the largest version of Bitcoin, we emulate three different sizes of slots, which we refer to as small, medium, and large. The small slot can fit a quarter-speed variant of Bitcoin, the medium slot can fit a half-speed variant, and the large slot can fit the full-speed variant. In the no-sharing approach, a single full-sized Bitcoin Morphlet is instantiated.

Figure 9 reports the full system runtime of each approach. When running only a single Bitcoin Morphlet, AMORPHOS is comparable to both the no sharing and two large slot approaches. The smaller slot-based approaches limit the size of the application and already perform worse than AMORPHOS. With two Bitcoin Morphlets, only

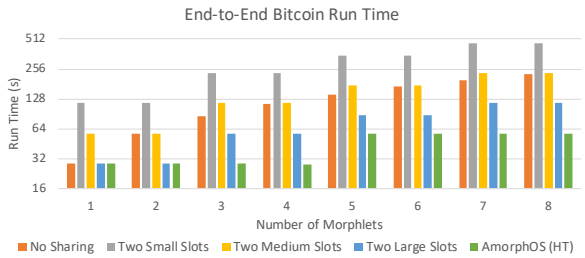


Figure 9: End-to-end runtime of Bitcoin executing under several different sharing schemes. Runtime is plotted logarithmically with lower runtimes being better.

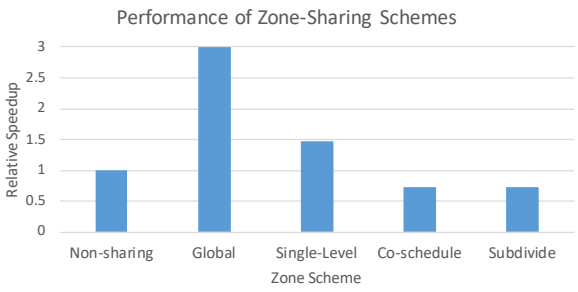


Figure 10: End-to-end performance of various zone-sharing schemes when executing 3 Bitcoin Morphlets.

AMORPHOS and two large slots are comparable. Finally, with 3 or more Bitcoin Morphlets, AMORPHOS is consistently able to attain higher logic densities and therefore better throughput than all other competing approaches. While AMORPHOS cannot always run in high-throughput mode as shown here, we expect AMORPHOS to maintain the same comparative advantage in the long run as it will only have to operate in low-latency mode until a high-throughput bitstream has been generated.

6.7 Hierarchical Zone Management

AMORPHOS can manage a zone in three ways. It can allocate the zone for exclusive use by a single Morphlet, co-schedule multiple Morphlets on it, or recursively subdivide it into two smaller zones. Subdividing top-level zones may be attractive if Morphlets do not fully utilize those zones or if Morphlet response time is more important than end-to-end run time. This flexibility gives rise to a policy space that trades off between density, performance, and registry overhead.

To characterize these trade-offs, we run three Bitcoin Morphlets on our Catapult prototype, in which AMORPHOS uses a single global zone or two top-level reconfigurable zones, each of which may be subdivided in two. We measure end-to-end execution time to completion for all three Morphlets, using a lower-bound baseline that does not share (**non-sharing**) and an upper-bound baseline that co-schedules all Morphlets on the global zone (**global**). We evaluate three different policies for managing the top-level zones. The first implements only a single-level of zone partitioning (**single-level**) with no co-scheduling within the zones. The second policy schedules combined Morphlets on zones without subdividing the zones (**co-schedule**). The third policy (**subdivide**) can subdivides the top-level zones. Registry entries for all combined bitstreams are pre-populated. For the **co-schedule** and **subdivide** cases, we morph the second two Morphlets by scaling them down to fit concurrently in a top-level reconfigurable zone, which reduces their throughput by a factor of 4, but allows us to run all three Morphlets in parallel.

Figure 10 shows end-to-end speedup relative to the non-sharing case for all policies. In this scenario, a single level of zone partitioning is the best option when co-scheduling on the global zone is not possible. This enables the first two Morphlets to run concurrently, providing additional concurrency that results in a performance gain relative to the no-sharing strategy. Both strategies for subdividing a top-level reconfigurable zone perform worse than the sequential case, for two reasons. First, performance is reduced by scaling them to fit a subdivided zone. Second, subdividing zones does not make all the underlying resources available to each subdivision. Additional PR logic is required for each, which consumes additional area and reduces routability.

Measurements of overhead for PR on Catapult FPGAs show that it grows linearly with density. Interconnect is the bottleneck resource, with 4% of global interconnect and 2% of global logic consumed by PR per Morphlet. While the 4% interconnect overhead can become a significant fraction of the allocatable fabric, density is primarily limited by fragmentation (we observe an average utilization loss of 16% in our workloads), which makes multiple levels of subdivision unprofitable for all but very small Morphlets on Catapult.

However, multiple levels of subdivision may be useful on F1 FPGAs, where the fraction of resources allocatable through PR zones is larger. F1 does not expose PR, so to predict sharing densities for PR-based subdivision on F1, we extrapolate assuming the same average 16% fragmentation per PR zone and 2% per-Morphlet overhead for PR logic. We assume that all fabric not consumed by AMORPHOS and PR logic can be divided evenly and allocated to Morphlets, but we impose a 90% upper bound on utilization per resource type, which is suggested by Xilinx to be the likely upper bound on UltraScale FPGAs [14]. This over-estimates utilizable fabric: other vendor guidance is more conservative [10] and our measured utilization does not exceed 70% for any workload.

Derived upper bounds on density for F1 hardware show that CLBs (Configurable Logic Blocks, which encapsulate multiple LUTs) are the limiting resource. We predict maximum sharing density with PR to be 16 and 4 for DNNWeaver and Bitcoin, respectively.[2] This suggests that zone subdivision will likely be possible and effective on F1. Our experience building AMORPHOS, however, is that subdividing zones increases the design complexity of hardware components and limits density unnecessarily by increasing fragmentation. In contrast, increasing density by co-scheduling Morphlets on a global zone can provide much higher densities with potentially higher effective deployment latency, but shifts much of the complexity to a software registry.

7 Related Work

FPGA programmability. Improving FPGA programmability is an active area largely characterized by efforts to enable programming with higher level languages, including C/C++ and other imperative languages [60, 19, 34, 69, 13, 18, 51, 68, 67], DSLs [32, 69, 95, 20, 81, 101, 66, 91], and even managed sequential languages such as C# [32] and map-reduce [95]. Progress in this area motivates our work, but is also orthogonal to it.

FPGA access to OS-managed resources. Prior work has explored exposing file systems [100] and the syscall interface [77, 100] to FPGAs. Much of this work has similar goals to our own, but we decided to focus on the exploration of cross-domain sharing and basic memory virtualization. A more mature AMORPHOS could clearly benefit from the rich body of work on memory virtualization for FPGAs [33, 15, 114, 77].

FPGA OSes. Previous work on FPGA OSes has focused on theoretical foundations for spatial sharing [43, 102, 108, 31], mechanisms for task preemption [73], relocation [55], context switch [72, 93], and scheduling of hardware and software tasks [25, 102, 108, 44]. While these explore ideas pertinent to OS primitives, end-to-end OS system-building was not their goal.

Extending current OS abstractions to FPGAs is another area of active research. ReconOS [77] extends a multi-threaded programming model to configurable SoCs that enables programmers to use "hardware threads" to transparently access OS-managed objects in the eCos [41] embedded OS. Hthreads [86] implements a similar hardware thread abstraction. Borph [100, 99] uses a *hardware process* abstraction to encapsulate FPGA logic in a process-like protection domain. Multi-application sharing for FPGAs is explored in [31, 109, 52], but some works restrict the programming model or design space [111], or do not tackle isolation and protection [31]. AMORPHOS differs by proposing new OS abstractions that differ from the existing CPU-oriented programming models.

MURAC [45] is the most closely related work to AMORPHOS. In MURAC, a process' logical address space encompasses all on-device resources that logically "belong to it", enabling the scheduler to support context switch using an ICAP (Internal Configuration Access Port). AMORPHOS takes a similar position on protection domains, but focuses on spatial scheduling and does not rely on hardware support for state capture.

FPGA Virtualization. Systems have been proposed that virtualize FPGAs with regions [88], tasks [89], processing elements [37], IPC-like communication primitives [80], and abstraction layers/overlays over diverse FPGA hardware [62, 50, 24, 61, 103]. Works virtualizing FPGAs in the cloud [30, 1, 79] share many of our core goals and tackle similar challenges. While these platforms use similar primitives to those of AMORPHOS, they typically restrict the programming and/or deployment model and do not support cross-domain sharing of FPGA fabric.

Overlays. FPGA overlays provide a virtualization layer to make a design independent of specific FPGA hardware [24, 113], enabling fast compilation times and low deployment latency [58, 64], at the cost of reduced hardware utilization and throughput. Like AMORPHOS, many overlays support some time-sharing and or spatial sharing. Overlays implement the same virtual architecture on different devices, they form a compatibility layer at the hardware interface. In contrast, AMORPHOS provides compatibility at the application-OS interface. Unlike Morphlets, overlays run on a virtual architecture, introducing overheads that limit utilization and performance.

8 Conclusion

This paper has described AMORPHOS, a design for FPGA protected sharing and compatibility based on abstractions that preserve existing programming models. AMORPHOS modulates between space- and time-sharing policies and isolates logic from different applications, enabling cross-cloud compatibility and dramatically improved throughput and utilization.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd Miguel Castro for their insights and comments. We acknowledge funding from NSF grant CNS-1618563.

[2] We do not predict density for MemDrive as it is bottlenecked by memory bandwidth at low densities.

References

[1] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. (Accessed on 09/27/2018).

[2] Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/. (Accessed on 04/30/2018).

[3] AMBA Specifications Arm. https://www.arm.com/products/system-ip/amba-specifications. (Accessed on 05/03/2018).

[4] AWS FPGA Marketplace. https://aws.amazon.com/marketplace/search/results?searchTerms=fpga. (Accessed on 09/14/2018).

[5] Catapult - Texas Advanced Computing Center. https://www.tacc.utexas.edu/systems/catapult/. (Accessed on 9/27/2018).

[6] Edico Genomes DRAGEN Platform. http://edicogenome.com/dragen-bioit-platform/. (Accessed on 5/2/2018).

[7] High-Level Synthesis with LegUp. http://legup.eecg.utoronto.ca/. (Accessed on 10/24/2017).

[8] Innova-2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. (Accessed on 5/2/2018).

[9] Live Video Encoding Using New AWS F1 Acceleration NGCodec. https://ngcodec.com/news/2017/3/31/live-video-encoding-using-new-aws-f1-acceleration. (Accessed on 09/27/2018).

[10] Measuring Device Performance and Utilization: A Competitive Overview (WP496). https://www.xilinx.com/support/documentation/white_papers/wp496-comp-perf-util.pdf. (Accessed on 09/27/2018).

[11] Microsoft unveils Project Brainwave for real-time AI - Microsoft Research. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/. (Accessed on 10/21/2017).

[12] programism/Open-Source-FPGA-Bitcoin-Miner. https://github.com/programism/Open-Source-FPGA-Bitcoin-Miner. (Accessed on 10/24/2017).

[13] SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. (Accessed on 09/27/2018).

[14] UltraScale Architecture: Highest Device Utilization, Performance, and Scalability (WP455). https://www.xilinx.com/support/documentation/white_papers/wp455-utilization.pdf. (Accessed on 09/27/2018).

[15] ADLER, M., FLEMING, K. E., PARASHAR, A., PELLAUER, M., AND EMER, J. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 25–28.

[16] ALTERA. Cyclone V SoC Development Board Reference Manual. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_cv_soc_dev_board.pdf. (Accessed on 5/2/2018).

[17] ALTERA. Integrating 100-GbE Switching Solutions on 28nm FPGAs. https://www.altera.com/en_US/pdfs/literature/wp/wp-01127-stxv-100gbe-switching.pdf. (Accessed on 5/2/2018).

[18] ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., SASS, R., AND ANDREWS, D. Enabling a uniform programming model across the software/hardware boundary. fccm'06, 2006.

[19] AUERBACH, J., BACON, D. F., CHENG, P., AND RABBAH, R. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 89–108.

[20] BACHRACH, J., VO, H., RICHARDS, B. C., LEE, Y., WATERMAN, A., AVIZIENIS, R., WAWRZYNEK, J., AND ASANOVIC, K. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012* (2012), pp. 1216–1225.

[21] BHASKER, J. *A Vhdl primer*. Prentice-Hall, 1999.

[22] BOHR, M. Moores Law Leadership. `https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/03/Mark-Bohr-2017-Moores-Law.pdf`. (Accessed on 09/27/2018).

[23] BOURGE, A., MULLER, O., AND ROUSSEAU, F. Automatic high-level hardware checkpoint selection for reconfigurable systems. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* (2015), IEEE, pp. 155–158.

[24] BRANT, A., AND LEMIEUX, G. G. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on* (2012), IEEE, pp. 93–96.

[25] BREBNER, G. J. A virtual hardware operating system for the xilinx xc6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers* (London, UK, UK, 1996), FPL '96, Springer-Verlag, pp. 327–336.

[26] BYMA, S., STEFFAN, J. G., BANNAZADEH, H., GARCIA, A. L., AND CHOW, P. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2014), FCCM '14, IEEE Computer Society, pp. 109–116.

[27] BYMA, S., TARAFDAR, N., XU, T., BANNAZADEH, H., LEON-GARCIA, A., AND CHOW, P. Expanding openflow capabilities with virtualized reconfigurable hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 94–97.

[28] CASPER, J., AND OLUKOTUN, K. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays* (New York, NY, USA, 2014), FPGA '14, ACM, pp. 151–160.

[29] CHAI, Z., YU, J., WANG, Z., ZHANG, J., AND ZHOU, H. An embedded fpga operating system optimized for vision computing (abstract only). In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 271–271.

[30] CHEN, F., SHAN, Y., ZHANG, Y., WANG, Y., FRANKE, H., CHANG, X., AND WANG, K. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (New York, NY, USA, 2014), CF '14, ACM, pp. 3:1–3:10.

[31] CHEN, L., MARCONI, T., AND MITRA, T. Online scheduling for multi-core shared reconfigurable fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2012), DATE '12, EDA Consortium, pp. 582–585.

[32] CHUNG, E. S., DAVIS, J. D., AND LEE, J. Linqits: Big data on little clients. In *40th International Symposium on Computer Architecture* (June 2013), ACM.

[33] CHUNG, E. S., HOE, J. C., AND MAI, K. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 97–106.

[34] COUSSY, P., AND MORAWIEC, A. *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media, 2008.

[35] CROCKETT, L. H., ELLIOT, R. A., ENDERWITZ, M. A., AND STEWART, R. W. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[36] DAI, G., CHI, Y., WANG, Y., AND YANG, H. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 105–110.

[37] DeHon, A., Markovsky, Y., Caspi, E., Chu, M., Huang, R., Perissakis, S., Pozzi, L., Yeh, J., and Wawrzynek, J. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems 30*, 6 (2006), 334–354.

[38] Dhar, S., Adya, S. N., Singhal, L., Iyer, M. A., and Pan, D. Z. Detailed placement for modern fpgas using 2d dynamic programming. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 9.

[39] Dhar, S., Iyer, M. A., Adya, S. N., Singhal, L., Rubanov, N., and Pan, D. Z. An effective timing-driven detailed placement algorithm for fpgas. In *Proceedings of the 2017 ACM on International Symposium on Physical Design, ISDP 2017, Portland, OR, USA, March 19-22, 2017* (2017), pp. 151–157.

[40] Dhar, S., and Pan, D. Z. Gdp: Gpu accelerated detailed placement. In *HPEC* (2018).

[41] Domahidi, A., Chu, E., and Boyd, S. Ecos: An socp solver for embedded systems. In *Control Conference (ECC), 2013 European* (2013), IEEE, pp. 3071–3076.

[42] Fahmy, S. A., Vipin, K., and Shreejith, S. Virtualized fpga accelerators for efficient cloud computing. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (Washington, DC, USA, 2015), CLOUDCOM '15, IEEE Computer Society, pp. 430–435.

[43] Fu, W., and Compton, K. Scheduling intervals for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on* (April 2008), pp. 87–96.

[44] Gonzalez, I., Lopez-Buedo, S., Sutter, G., Sanchez-Roman, D., Gomez-Arribas, F. J., and Aracil, J. Virtualization of reconfigurable coprocessors in hprc systems with multicore architecture. *J. Syst. Archit. 58*, 6-7 (June 2012), 247–256.

[45] Hamilton, B. K., Inggs, M., and So, H. K. H. Scheduling mixed-architecture processes in tightly coupled fpga-cpu reconfigurable computers. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on* (May 2014), pp. 240–240.

[46] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[47] Hansen, S. G., Koch, D., and Torresen, J. High speed partial run-time reconfiguration using enhanced icap hard macro. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (2011), IEEE, pp. 174–180.

[48] Hara, Y., Tomiyama, H., Honda, S., Takada, H., and Ishii, K. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on* (2008), IEEE, pp. 1192–1195.

[49] Hari Krishnan, R., and Sai Saketh, Y. Cryptocurrency mining–transition to cloud.

[50] Huang, C.-H., and Hsiung, P.-A. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embed. Syst. Lett. 1*, 1 (May 2009), 19–23.

[51] Inc, S. C. Carte programming environment, 2006.

[52] Ismail, A., and Shannon, L. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2011), FCCM '11, IEEE Computer Society, pp. 170–177.

[53] István, Z., Sidler, D., Alonso, G., and Vukolic, M. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 425–438.

[54] Kaganov, A., Lakhany, A., and Chow, P. Fpga acceleration of multifactor cdo pricing. *ACM Trans. Reconfigurable Technol. Syst. 4*, 2 (May 2011), 20:1–20:17.

[55] KALTE, H., AND PORRMANN, M. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on* (Aug 2005), pp. 223–228.

[56] KALTE, H., AND PORRMANN, M. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on* (2005), IEEE, pp. 223–228.

[57] KAPITZA, R., BEHL, J., CACHIN, C., DISTLER, T., KUHNLE, S., MOHAMMADI, S. V., SCHRÖDER-PREIKSCHAT, W., AND STENGEL, K. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 295–308.

[58] KAPRE, N., AND GRAY, J. Hoplite: Building austere overlay nocs for fpgas. In *FPL* (2015), IEEE, pp. 1–8.

[59] KARA, K., AND ALONSO, G. Fast and robust hashing for database operators. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016* (2016), pp. 1–4.

[60] KHRONOS GROUP. *The OpenCL Specification, Version 1.0*, 2009.

[61] KIRCHGESSNER, R., GEORGE, A. D., AND STITT, G. Low-overhead fpga middleware for application portability and productivity. *ACM Trans. Reconfigurable Technol. Syst. 8*, 4 (Sept. 2015), 21:1–21:22.

[62] KIRCHGESSNER, R., STITT, G., GEORGE, A., AND LAM, H. Virtualrc: A virtual fpga platform for applications and tools portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2012), FPGA '12, ACM, pp. 205–208.

[63] KNODEL, O., AND SPALLEK, R. G. RC3E: provision and management of reconfigurable hardware accelerators in a cloud environment. *CoRR abs/1508.06843* (2015).

[64] KOCH, D., BECKHOFF, C., AND LEMIEUX, G. G. F. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL* (2013), IEEE, pp. 1–8.

[65] KOEPLINGER, D., DELIMITROU, C., PRABHAKAR, R., KOZYRAKIS, C., ZHANG, Y., AND OLUKOTUN, K. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 115–127.

[66] KOEPLINGER, D., DELIMITROU, C., PRABHAKAR, R., KOZYRAKIS, C., ZHANG, Y., AND OLUKOTUN, K. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 115–127.

[67] KOEPLINGER, D., FELDMAN, M., PRABHAKAR, R., ZHANG, Y., HADJIS, S., FISZEL, R., ZHAO, T., NARDI, L., PEDRAM, A., KOZYRAKIS, C., AND OLUKOTUN, K. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, ACM, pp. 296–311.

[68] LEBAK, J., KEPNER, J., HOFFMANN, H., AND RUTLEDGE, E. Parallel vsipl++: An open standard software library for high-performance parallel signal processing. *Proceedings of the IEEE 93*, 2 (2005), 313–330.

[69] LEBEDEV, I. A., FLETCHER, C. W., CHENG, S., MARTIN, J., DOUPNIK, A., BURKE, D., LIN, M., AND WAWRZYNEK, J. Exploring many-core design templates for fpgas and asics. *Int. J. Reconfig. Comp. 2012* (2012), 439141:1–439141:15.

[70] LEBER, C., GEIB, B., AND LITZ, H. High frequency trading acceleration using fpgas. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications* (Washington, DC, USA, 2011), FPL '11, IEEE Computer Society, pp. 317–322.

[71] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[72] LEE, T.-Y., HU, C.-C., LAI, L.-W., AND TSAI, C.-C. Hardware context-switch methodology for dynamically partially reconfigurable systems. *J. Inf. Sci. Eng. 26* (2010), 1289–1305.

[73] LEVINSON, L., MANNER, R., SESSLER, M., AND SIMMLER, H. Preemptive multitasking on fpgas. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on* (2000), pp. 301–302.

[74] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 476–488.

[75] LIN, E. C., AND RUTENBAR, R. A. A multi-fpga 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (2009), ACM, pp. 83–92.

[76] LIU, M., KUEHN, W., LU, Z., AND JANTSCH, A. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* (2009), IEEE, pp. 498–502.

[77] LÜBBERS, E., AND PLATZNER, M. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst. 9*, 1 (Oct. 2009), 8:1–8:33.

[78] LYSECKY, R., MILLER, K., VAHID, F., AND VISSERS, K. Firm-core virtual fpga for just-in-time fpga compilation (abstract only). In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays* (New York, NY, USA, 2005), FPGA '05, ACM, pp. 271–271.

[79] MICROSOFT. Microsoft azure goes back to rack servers with project olympus, 2017.

[80] MISHRA, M., CALLAHAN, T. J., CHELCEA, T., VENKATARAMANI, G., GOLDSTEIN, S. C., AND BUDIU, M. Tartan: Evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev. 40*, 5 (Oct. 2006), 163–174.

[81] MOORE, N., CONTI, A., LEESER, M., CORDES, B., AND KING, L. S. An extensible framework for application portability between reconfigurable supercomputing architectures, 2007.

[82] MURALIDHARAN, S., O'BRIEN, K., AND LALANNE, C. A semi-automated tool flow for roofline anaylsis of opencl kernels on accelerators. In *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)* (2015).

[83] NURVITADHI, E., VENKATESH, G., SIM, J., MARR, D., HUANG, R., HOCK, J. O. G., LIEW, Y. T., SRIVATSAN, K., MOSS, D., SUBHASCHANDRA, S., ET AL. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *FPGA* (2017), pp. 5–14.

[84] OCTOPART. Octopart historical pricing, 2017.

[85] OGUNTEBI, T., AND OLUKOTUN, K. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 111–117.

[86] PECK, W., ANDERSON, E. K., AGRON, J., STEVENS, J., BAIJOT, F., AND ANDREWS, D. L. Hthreads: A computational model for reconfigurable devices. In *FPL* (2006), IEEE, pp. 1–4.

[87] PELLERIN, D. Accelerated Computing on AWS. `http://asapconference.org/slides/amazon.pdf`, July 2017. (Accessed on 5/2/2018).

[88] PHAM, K. D., JAIN, A. K., CUI, J., FAHMY, S. A., AND MASKELL, D. L. Microkernel hypervisor for a hybrid arm-fpga platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on* (June 2013), pp. 219–226.

[89] PLESSL, C., AND PLATZNER, M. Zippy-a coarse-grained reconfigurable array with support for hardware virtualization. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on* (2005), IEEE, pp. 213–218.

[90] PRABHAKAR, R., KOEPLINGER, D., BROWN, K. J., LEE, H., DE SA, C., KOZYRAKIS, C., AND OLUKOTUN, K. Generating configurable hardware from parallel patterns. *SIGOPS Oper. Syst. Rev. 50*, 2 (Mar. 2016), 651–665.

[91] PRABHAKAR, R., ZHANG, Y., KOEPLINGER, D., FELDMAN, M., ZHAO, T., HADJIS, S., PEDRAM, A., KOZYRAKIS, C., AND OLUKOTUN, K. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 389–402.

[92] PUTNAM, A., CAULFIELD, A., CHUNG, E., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)* (June 2014).

[93] RUPNOW, K., FU, W., AND COMPTON, K. Block, drop or roll(back): Alternative preemption methods for RH multi-tasking. In *FCCM 2009, 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, USA, 5-7 April 2009, Proceedings* (2009), pp. 63–70.

[94] SHAFIEE, A., GUNDU, A., SHEVGOOR, M., BALASUBRAMONIAN, R., AND TIWARI, M. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, 2015), MICRO-48, ACM, pp. 89–101.

[95] SHAN, Y., WANG, B., YAN, J., WANG, Y., XU, N.-Y., AND YANG, H. Fpmr: Mapreduce framework on fpga. In *FPGA* (2010), P. Y. K. Cheung and J. Wawrzynek, Eds., ACM, pp. 93–102.

[96] SHARMA, H., PARK, J., AMARO, E., THWAITES, B., KOTHA, P., GUPTA, A., KIM, J. K., MISHRA, A., AND ESMAEILZADEH, H. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures* (2016).

[97] SIDLER, D., ISTVÁN, Z., OWAIDA, M., AND ALONSO, G. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 403–415.

[98] SIMMLER, H., LEVINSON, L., AND MÄNNER, R. Multitasking on fpga coprocessors. *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing* (2000), 121–130.

[99] SO, H. K.-H., AND BRODERSEN, R. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Trans. Embed. Comput. Syst. 7*, 2 (Jan. 2008), 14:1–14:28.

[100] SO, H. K.-H., AND BRODERSEN, R. W. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2007.

[101] SO, H. K.-H., AND WAWRZYNEK, J. Olaf'16: Second international workshop on overlay architectures for fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 1–1.

[102] STEIGER, C., WALDER, H., AND PLATZNER, M. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers 53*, 11 (Nov 2004), 1393–1407.

[103] STITT, G., AND COOLE, J. Intermediate fabrics: Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters 3*, 3 (Sept 2011), 81–84.

[104] SUDA, N., CHANDRA, V., DASIKA, G., MOHANTY, A., MA, Y., VRUDHULA, S., SEO, J.-S., AND CAO, Y. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 16–25.

[105] TAYLOR, M. B. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2013), IEEE Press, p. 16.

[106] THOMAS, D., AND MOORBY, P. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[107] TSUTSUI, A., MIYAZAKI, T., YAMADA, K., AND OHTA, N. Special purpose fpga for high-speed digital telecommunication systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors* (Washington, DC, USA, 1995), ICCD '95, IEEE Computer Society, pp. 486–491.

[108] WASSI, G., BENKHELIFA, M. E. A., LAWDAY, G., VERDIER, F., AND GARCIA, S. Multi-shape tasks scheduling for online multitasking on fpgas. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on* (May 2014), pp. 1–7.

[109] WATKINS, M. A., AND ALBONESI, D. H. Remap: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 497–508.

[110] WEERASINGHE, J., ABEL, F., HAGLEITNER, C., AND HERKERSDORF, A. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Beijing, China, August 10-14, 2015* (2015), pp. 1078–1086.

[111] WERNER, S., OEY, O., GÖHRINGER, D., HÜBNER, M., AND BECKER, J. Virtualized on-chip distributed computing for heterogeneous reconfigurable multi-core systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2012), DATE '12, EDA Consortium, pp. 280–283.

[112] WEST, B., CHAMBERLAIN, R. D., INDECK, R. S., AND ZHANG, Q. An fpga-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors* (2003), vol. 12, pp. 25–32.

[113] WIERSEMA, T., BOCKHORN, A., AND PLATZNER, M. Embedding FPGA overlays into configurable systems-on-chip: Reconos meets ZUMA. In *ReConFig* (2014), IEEE, pp. 1–6.

[114] WINTERSTEIN, F., FLEMING, K., YANG, H.-J., BAYLISS, S., AND CONSTANTINIDES, G. Matchup: Memory abstractions for heap manipulating programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 136–145.

[115] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 161–170.

[116] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015), ACM, pp. 161–170.

[117] ZHAO, R., SONG, W., ZHANG, W., XING, T., LIN, J.-H., SRIVASTAVA, M. B., GUPTA, R., AND ZHANG, Z. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA* (2017), pp. 15–24.

Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing

Kiwan Maeng Brandon Lucia
Carnegie Mellon University
{*kmaeng,blucia*}*@andrew.cmu.edu*
http://intermittent.systems

Abstract

Energy-harvesting devices have the potential to be the foundation of emerging, sensor-rich application domains where the use of batteries is infeasible, such as in space and civil infrastructure. Programming on an energy-harvesting device is difficult because the device operates only intermittently, as energy is available. Intermittent operation requires the programmer to reason about energy to understand data consistency and forward progress of their program. Energy varies with input and environment, making intermittent programming difficult. Existing systems for intermittent execution provide an unfamiliar programming abstraction and fail to adapt to energy changes forcing a compromise of either performance or assurance of forward progress.

This paper presents Chinchilla, a compiler and runtime system that allows running unmodified C code efficiently on an energy-harvesting device with little additional programmer effort and no additional hardware support. Chinchilla overprovisions code with checkpoints to assure the system makes progress, even with scarce energy. Chinchilla disables checkpoints dynamically to efficiently adapt to energy conditions. Experiments show that Chinchilla improves programmability, is performant, and makes it simple to statically check the absence of non-termination. Comparing to two systems from prior work, Alpaca and Ratchet, Chinchilla makes progress when Alpaca cannot, and has 125% mean speedup against Ratchet.

1 Introduction

The maturation of energy-harvesting technology and low-power microcontrollers fostered batteryless devices that operate using energy from their environment. *Energy-harvesting devices* operate by collecting and buffering energy in a capacitor, and only *intermittently* executing the software when there is available energy in the capacitor. During execution, a device computes, uses volatile and non-volatile (e.g., FRAM [50]) memory, reads sensors and communicates. Recent work enabled intermittent software execution [30]. Some capture checkpoints [36, 44, 51] automatically at arbitrary points to make progress despite power failures. Other work asks the programmer to decompose code into idempotent, atomic tasks [12, 31, 35] that attempt to execute repeatedly until completing uninterrupted.

While successful enablers of intermittent computing, these prior systems compromise on one or more important system design aspects: performance, programmability, and avoidance of non-termination. Automatic checkpointing approaches [36,44,51] make programming simple, but often incur a high execution time overhead due to excessive checkpoints. Explicit task models [12, 31, 35] require the programmer to adhere to the task-based programming model, hampering programmability. Checkpointing and task-based systems also do not provide a simple way of checking whether the system can encounter non-termination. The code between two consecutive checkpoints or within a task (we refer to both as "task execution") may require more energy to complete than will ever be available in the device's fixed energy buffer. When a task's execution consumes more energy than hardware can buffer, the task will not execute to completion, and the system faces *non-termination* — repeated attempts to execute a task that will never complete. Even a hard reboot does not recover from non-termination, because intermittent operation spans power failures. The only fix is to change the code to use smaller tasks or add more frequent checkpoints and re-flashing the code onto the device.

Avoiding non-termination is an important correctness property in an intermittent system that no prior system satisfactorily provides. Task-based systems [12, 31, 35] complicate programming by asking the programmer to estimate the energy use of code regions and to divide code into arbitrary tasks that execute atomically. Checkpoint-based systems [36, 44, 51] place checkpoint at semantically meaningful points in a program, oblivious to energy consumption between checkpoints. Energy consumption may vary with input and the environment, and static energy modeling [4,7,13] is often imprecise and overly conservative. Without a way of reliably checking a program for non-termination, the burden of writing performant, correct applications lies entirely on the programmer in existing intermittent systems.

In this work, we observe that *adaptation* of checkpoint timing based on energy consumption is the key to achieve the three goals: high performance, accessi-

ble programmability, and a simple static check for the absence of non-termination. Based on this observation, we propose Chinchilla, [1] a checkpointing, task-based runtime system that dynamically adapts the interval between checkpoints based on direct observations of program progress. Chinchilla does not ask the programmer to specify task boundaries, making programming simple. Chinchilla statically overprovisions a program with potential checkpoints and makes it simple to check that the span between two potential checkpoints will not exceed the device's energy capacity. Chinchilla achieves high performance by dynamically adapting which potential checkpoints to collect, based on the program's rate of progress, which will vary across platforms and inputs.

We implemented a full prototype of Chinchilla including compiler support, a runtime system, and a non-termination checker. We evaluated Chinchilla running on several real RF-harvesting hardware setups, running a collection of programs from the literature [19, 35] and comparing to two representative systems from prior work, Alpaca and Ratchet [35, 51]. We show that Chinchilla improves programmability, supporting most of the C language (including libraries) and avoids re-engineering code for every new platform. Chinchilla achieves high performance, with a 2.25x average speedup compared to Ratchet, the previous state-of-the-art automatic checkpointing system. Chinchilla achieves performance parity with the task-based Alpaca that is faster than Ratchet, but requires substantial code re-engineering. Additionally, we show that Chinchilla makes it simple to check for the absence of non-termination, providing an assurance that code will run correctly once deployed.

In summary, Chinchilla's main features are:

- A substantial performance improvement compared to state-of-the-art intermittent checkpointing systems.
- Enabling a simple, static check that gives assurance that a program avoids non-termination.
- A simple programming model that supports most of the C language.
- A dynamic run-time checkpointing adaptation mechanism that accommodates varied inputs and environmental conditions.

2 Background

Energy-harvesting devices extract energy from their environment and execute software according to the intermittent execution model, which presents unique challenges that are not present under continuous execution.

[1] **C**orrect, **H**ardware-agnostic **IN**termittent **CH**eckpointing **I**nstrumentation **L**ayer with **L**ow-overhead **A**daptation

2.1 Energy-Harvesting Devices

An energy-harvesting device includes a microcontroller (MCU), sensors, volatile and non-volatile memory, and radios. An energy-harvesting device extracts energy from its environment, e.g., radio, vibration, or light, and operates *intermittently*, only when energy is available. A device collects energy into a fixed size energy buffer, usually a capacitor. While the device is inactive, energy slowly accumulates in the buffer. When the energy level in the buffer reaches a defined threshold, the device operates, quickly consuming the buffered energy. The time to accumulate energy is usually greater by orders of magnitude than the time to consume the energy. For example, a WISP with a nearby RF power supply may charge for a second to support 10 ms of operation [14, 47]. At a failure, the device loses the contents of its registers, volatile memory, and peripheral configuration, while retaining the contents of its non-volatile memory (e.g., FRAM [50]).

2.2 Correctness in Intermittent Execution

Software on an energy-harvesting device executes according to the *intermittent execution model*. After a power failure, control resumes from some prior point and execution continues instead of terminating. Key challenges of intermittent execution are: ensuring (1) memory consistency, and (2) forward progress.

Figure 1 shows the challenges of intermittent execution. The figure shows a code for a 1-D convolution that preserves execution progress on each power failure by collecting a volatile execution context (registers, stack) on each outer loop iteration (a model similar to Mementos [44]). The `out` array is allocated in non-volatile memory, initialized to zero. The two executions in the figure show two problematic intermittent execution behaviors. Execution 1 shows that if power fails after updating `out[0]` but without reaching the checkpoint, control flow reverts to the top of the inner loop (`j = 0`) on reboot. However, the partially updated value of `out[0]` persists after the power failure. On reboot, the code updates `out[0]` again, leading to a memory state that is impossible in a continuously-powered execution. Execution 2 shows that if the inner loop's bound, `K`, is sufficiently large, the system will exhaust its energy before reaching the checkpoint, leading to a non-termination.

Several prior strategies successfully ensure memory consistency on intermittent execution, i.e. they solved the problem from Execution 1. However, they show limitations in avoiding non-termination (problem from Execution 2). We discuss how two popular previous approaches — explicit task-based models and automatic checkpointing systems — try to avoid non-termination on intermittent execution and what their limitations are.

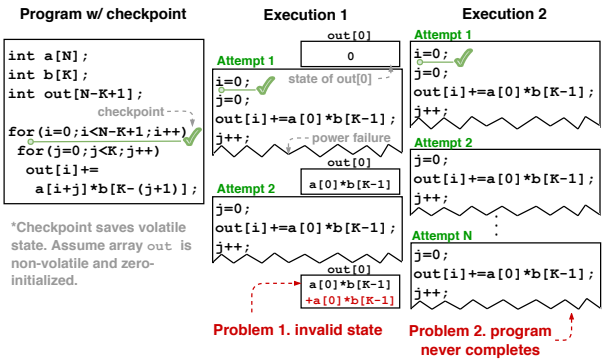
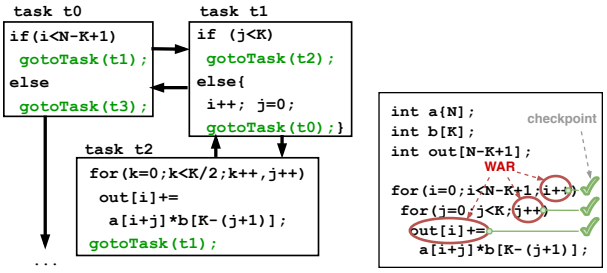


Figure 1: **Challenges of intermittent execution.** Code with volatile state checkpoints may leave memory inconsistent (Execution 1) or never terminate (Execution 2).



(a) Explicit task model

(b) Automatic checkpointing system

Figure 2: **Different systems for the convolution.** A convolution code written in (a) explicit task model (Alpaca [35]), and a code generated by (b) automatic checkpointing system (Ratchet [51]).

Explicit Task Models Explicit task-based intermittent programming and execution models *require* the programmer to explicitly specify task boundaries [12, 31, 35]. In these models, it is solely the programmer's responsibility to avoid non-termination. These models require careful programming, because if a task consumes more energy than the device can buffer, the task will enter non-termination. The programmability cost of specifying task boundaries is high, especially because estimating the energy use of a task for various inputs is difficult.

Automatic Checkpointing Systems Automatic checkpointing systems statically insert a checkpoint at arbitrary program points using compiler and runtime support [36, 44, 51]. Most of these systems insert a large number of checkpoints throughout the binary without considering whether there are sufficiently frequent checkpoints to avoid non-termination. Excessive frequent checkpoints can have high overhead, and there is no easy way in such a system to statically check the presence of non-termination. Additionally, automatic checkpointing systems do not allow the programmer to control over the duration and energy consumption of a task. If task energy demand exceeds the device energy supply, the programmer has no recourse to fix the issue, because checkpoint placement is not part of the programming model. Some propose *ad hoc*, dynamic fallbacks that can have high overhead, and are difficult to characterize [51].

Some recent systems tried to estimate task energy cost and place checkpoints accordingly instead of heuristically. However, prior work showed that precisely estimating the energy cost for an arbitrary code is a challenge even when restricting the model only to the MCU core instead of the full system [9,29]. Models that rely on the instruction counting as a proxy for energy [4] or that use statistical energy models [13] are useful, but limited in precision. To make non-termination checking simple, Chinchilla should have a static check that accounts for full-system power and does not rely on proxy measurements or statistical models.

2.3 Programmability and Performance in Existing Models

In addition to non-termination, prior systems may make programming complex or have poor performance. Figure 2 shows how prior task and checkpointing systems may have programmability and performance issues.

Explicit Task Models Programming with explicit tasks is difficult. Figure 2a shows how the programmer could write a 1-D convolution code in a task-based model [12, 35]. The syntax deviates from plain C and requires the programmer to decide how many loop iterations fit in a task without exceeding the device's energy budget (e.g., task `t2` is chosen to hold `K/2` loop iterations). A bad choice that puts too many iterations in a task leads to non-termination. A different bad choice that puts too few iterations in a task sacrifices performance. Crucially, if the energy buffer or input changes, the programmer has to *re-write the code* to make tasks differently. Recent platforms support a dynamically variable energy buffer size [14], making the problem more urgent.

Automatic Checkpointing Systems Although automatic checkpoint insertion to the binary incurs low to no additional programming effort, an excess of checkpoints may lead to high execution time overhead. As shown in Figure 2b, Ratchet [51], the state-of-the-art automatic checkpointing system, inserts checkpoint between every Write-After-Read (WAR) dependence, possibly inserting *two* checkpoints on each inner loop iteration and one checkpoint on each outer loop iteration in the example.

If the system's energy buffer can complete multiple iterations of the loop without a power failure, Ratchet suffers an unnecessarily high overhead. Ratchet cannot selectively skip a checkpoint because checkpointing is required by Ratchet's memory consistency model.

2.4 Task Atomicity

An automatic checkpointing system also fails to provide a mechanism for specifying and enforcing application-level atomicity constraints on checkpoint placement. For example, if a program should access two sensors atomically at the same time, they should not be interleaved by a checkpoint. If a checkpoint splits the atomic region, the value collected by the first sensor may be from before a power interruption, and the value collected from the second sensor access may be from much later, after the power interruption, resulting in stale data. Chinchilla allows the programmer to specify such atomicity constraints if an application needs them.

3 System Overview

Chinchilla is a software system that uses a novel, adaptive checkpointing scheme to make software on an intermittently-operating system execute correctly and efficiently. Chinchilla statically overprovisions code with potential checkpoints and dynamically deactivates unnecessary checkpoints at run time to minimize performance overhead. Chinchilla is designed to improve the *programmability* and *efficiency* of intermittent systems, while avoiding *non-termination*. Programming is easy, because Chinchilla inserts checkpoints automatically. Execution is efficient, because Chinchilla's dynamic adaptation mechanism minimizes its checkpointing and state management overhead. Chinchilla exposes a simple, statically-checkable property to determine whether a program will behave correctly on a given platform, allowing Chinchilla to avoid non-termination and effectively making Chinchilla portable to systems with a wide range of energy buffering capacities.

Figure 3 provides an overview of Chinchilla's main features, which are implemented in an instrumenting compiler analysis and software runtime system. Chinchilla compiler inserts checkpointing instrumentation that captures registers and part of the non-volatile data. The compiler also uses static analysis to detect which *protected* data must persist across checkpoints and power failures. Chinchilla's runtime system implements checkpoint and restart, a non-volatile stack to avoid full stack checkpointing, dynamic support for selectively activating checkpoints, and undo logging to ensure consistency.

First, we discuss *where* the compiler inserts checkpoints, second we describe *what* data are checkpointed and logged, and third, we describe *how* Chinchilla selectively activates checkpoints to mitigate overheads.

3.1 Placing Checkpoints to Enable Static Non-Termination Checks

Chinchilla inserts checkpoints into a program that preserve its progress by saving execution context that Chinchilla can restore after a power failure. Chinchilla's goals in placing checkpoints, are to preserve progress and avoid non-termination, and to minimize run time overhead. These goals are in tension. To avoid non-termination, Chinchilla must insert a checkpoint along any program path that consumes more energy than the system can buffer, conservatively checkpointing *as frequently as possible* to avoid non-termination with an arbitrarily small energy buffer. To minimize checkpoint overheads, Chinchilla should only checkpoint at the *boundaries* of a path that consumes more energy than the device can buffer, ideally checkpointing *as infrequently as possible*. Compounding the problem, measuring the full-system energy consumption of arbitrary code is challenging and imprecise [13, 29] because path energy depends on inputs and peripheral state.

Chinchilla escapes the checkpoint placement dilemma by inserting checkpoints conservatively into the program so that the resulting program can be simply assured to avoid non-termination, and selectively disabling checkpoints that are unnecessary to minimize overhead. Chinchilla inserts a checkpoint at each boundary of arbitrarily-defined spans of the program, which we refer to as *checkpoint blocks*. A checkpoint block defines the minimum span of code after which a checkpoint might occur — Chinchilla inserts checkpoints at the boundaries of checkpoint blocks, but not inside a block. If no checkpoint block consumes more energy than the device can buffer, then the program will not suffer non-termination. Given this *block energy sufficiency* premise, eventually every checkpoint block will complete, reaching the next checkpoint, and preserving its progress.

The effectiveness of Chinchilla relies on a well-chosen definition of a checkpoint block. A well-chosen block definition is easily identifiable statically, permits frequent block boundaries, allows easily measuring block energy cost, and yields blocks with low energy variance. Statically identifying block extents is important for statically enumerating all possible program control-flow behavior, especially in the presence of complex I/O. Block boundaries must naturally occur frequently enough in a program, or must be easy to insert arbitrarily frequently to ensure block energy sufficiency, even with a small energy buffer. A block's energy should be easy to measure and have low variance, which precludes any block definition that has unbounded loops or input-dependent control-flow paths with wildly different energy costs.

While many block definitions may fit these requirements, Chinchilla uses the basic block as its checkpoint block definition because it fits the criteria well. Basic

blocks are statically defined, frequently occurring, and can be arbitrarily subdivided by a compiler as needed to suit small energy buffers. Basic blocks do not contain branches precluding loops and input-dependent paths, which may vary substantially in energy consumption.

Some multi-basic-block regions of code must be atomic and cannot be spliced by a checkpoint, such as code that reads, processes, and records values from related sensors (cf. Section 2.4). The programmer can annotate such code as an `atomic` block, and Chinchilla will treat it as a single checkpoint block. The programmer must manually ensure that such an `atomic` block meets the criteria of a well-chosen checkpoint block. Annotation of the `atomic` blocks is also a feature of Chinchilla that previous compiler-based systems neglected [44, 51].

Checking for Non-Termination. While Chinchilla compiler itself does not provide a static termination guarantee, it makes *checking* for non-termination simple: if no block's energy consumption exceeds the device's energy buffer, the program avoids non-termination. A programmer can check for non-termination by measuring basic block energy consumption under exhaustive, randomized, or representative inputs. In this work, Chinchilla adapts the CleanCut energy-measuring compiler's block measurement tool [13] to check block energy, exposing the checker directly to the programmer. After the compiler instruments each checkpoint block and the programmer checked that no block's energy consumption leads to non-termination using the checker, the program is *safe*, but *over-provisioned* with checkpoints. Section 3.3 describes how Chinchilla selectively disables checkpoints to avoid excessive overheads.

Limitations of Chinchilla's Assurance of Non-Termination. Even with Chinchilla's compiler and checker, Chinchilla cannot always guarantee the absence of non-termination due to possible variation in energy consumption with variation in input and environment. Despite this limitation, Chinchilla provides two major advantages. First, Chinchilla shifts the scope of reasoning about non-termination from arbitrary inter-checkpoint code regions to a single basic block. Single basic blocks have a lower variance in their energy consumption, simplifying energy measurement and non-termination reasoning [13, 29]. Second, Chinchilla selectively disables unnecessary checkpoints allowing for conservative, static over-provisioning with checkpoints (i.e., on every block). Leveraging these properties Chinchilla provides improved assurance of non-termination (although not a guarantee of its absence in all conditions). Practically, Chinchilla eliminates non-termination. Our evaluation shows that Chinchilla's conservative over-provisioning with checkpoints leaves a 2,100% margin between the device's energy capacity and the highest energy cost of any block; even extreme variation in block energy cost due to inputs or environment is unlikely to exceed such a large margin and cause non-termination (Figure 8).

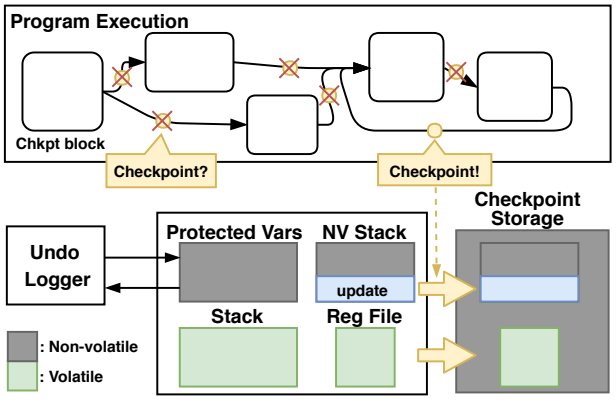


Figure 3: **Overview of Chinchilla.**

3.2 Checkpointing and Undo Logging

Chinchilla checkpoints execution context to preserve and uses undo logging to keep selected, protected non-volatile data consistent across failures.

Checkpointing. Chinchilla checkpoints the execution context, consisting of just the register file and part of the non-volatile data, but not the stack or global data, making the time and energy cost of checkpointing small and predictable. Chinchilla is unlike prior work that uses a fully non-volatile stack (e.g., [25, 45, 51]) to afford register-only checkpointing. Instead, Chinchilla uses an efficient volatile stack and *promotes* a subset of variables to reside in non-volatile memory. Chinchilla only promotes data that may not be re-initialized after a checkpoint to non-volatile memory, leaving all other data on the volatile stack. Chinchilla compiler uses a live-range analysis [3] to identify stack data to promote. If a variable's live range begins after a checkpoint, the variable will be assigned before it is read after a power interruption. Such a variable is safe to leave on the volatile stack without additional protection. The data which need promotion but are not visible to the compiler pass (e.g, data generated by the latter stage of the compiler) is handled by our non-volatile stack discussed below.

Undo Logging. Chinchilla keeps compiler-selected protected, non-volatile variables consistent using undo logging. The key problem, as prior work [12, 25, 31, 35, 43, 51] observed, is that if a non-volatile memory access is involved in a write-after-read (WAR) dependence, then an update to the variable during an execution attempt before a power interruption may incorrectly be visible to a re-executed read after the power interruption.

To prevent code from reading incorrect values, Chinchilla instruments each *write* to a protected variable with

undo logging code. At run time, the undo logging code saves to a log the value of the protected variable before the variable's first write after a checkpoint. Chinchilla rolls back updates to protected variables before restarting execution after a power interruption using the log. Section 4.3 describes our undo logging implementation.

Non-Volatile Stack Data. Chinchilla uses a small non-volatile stack to persist stack data that are not visible to Chinchilla compiler pass. These data include return addresses and spilled registers. Compared to the stack, which may be kilobytes, the non-volatile stack is typically small ($\sim$10 bytes) and its elements short-lived. Section 4.2 describes the compiler back-end and runtime system for the non-volatile stack.

3.3 Selective Checkpointing

Checkpointing on every basic block would have a high run time cost that is usually unnecessary because a system is unlikely to fail on every basic block. Chinchilla mitigates the cost of its non-termination-avoiding, conservative, provisioning of checkpoints by skipping some checkpoints at execution time.

Chinchilla sets a timer at startup that, upon its expiration, indicates that Chinchilla should collect the next dynamically executed checkpoint. The runtime skips any checkpoint it encounters while the timer is running, i.e., before it elapses. The key challenge for Chinchilla is identifying a timer duration that expires before the device exhausts its buffered energy (ideally checkpointing before failing), but does not expire too frequently (ideally checkpointing only just before failing).

Chinchilla binary searches for an ideal timer interval at runtime. Chinchilla's search starts by running with a long timer interval. If power fails before the timer expires and Chinchilla collects no checkpoint, the interval is too long; Chinchilla halves the interval and tries again. Assuming that no block consumes more energy than the device can buffer, the timer duration eventually decreases sufficiently to reach a checkpoint.

After finding a sufficiently short interval Chinchilla tries to avoid excessively frequent checkpointing by opportunistically increasing the interval again. Given a *new* shorter interval and the *old* longer interval, Chinchilla increases the interval to a new *median* interval halfway between the new and old intervals. Chinchilla increases its interval to the median interval *only if* execution continues past the new median interval and successfully captures a checkpoint. While non-termination requires immediate interval adjustment, increasing the interval is less urgent. Chinchilla allows the user to decide when in the code to update intervals (e.g., every 100 reboots, each outer loop) by manually annotating a *tuning point*. We put a tuning point on the outermost loop in our benchmarks.

4 Chinchilla Implementation

We implemented a prototype of Chinchilla with four parts: an instrumenting compiler pass and back-end, a runtime library, and a block non-termination checker.

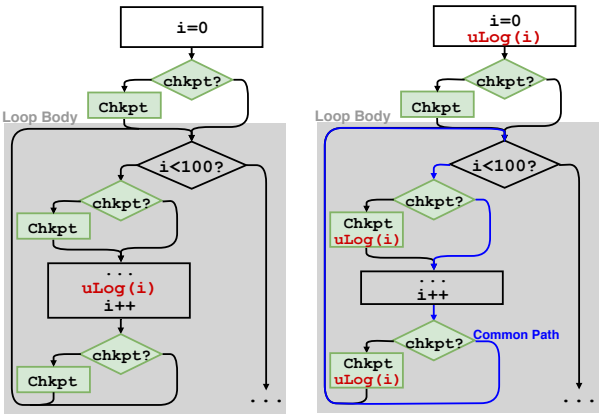
4.1 The Chinchilla Compiler

Chinchilla's compiler transforms C code to use the Chinchilla runtime for safe intermittent execution. The compiler performs five transformations on the code. First, the compiler adds checkpoints at the entry of each basic block. Second, the compiler uses live variable analysis to identify variables that need protection. Third, Chinchilla adds undo logging instrumentation to writes to protected variables. Fourth, the compiler lays out protected variables in memory to efficiently support metadata. Fifth, Chinchilla re-writes `main()` to re-initialize peripherals and roll back the undo logs on reboot.

Checkpoint Instrumentation. The Chinchilla compiler inserts checkpoint code between every pair of basic blocks, except for blocks in explicitly annotated atomic regions. The checkpoint code checks a flag maintained by the Chinchilla runtime that indicates whether the checkpoint interval has elapsed since the last power failure. When the flag is set, the interval has elapsed and the checkpoint code captures a checkpoint.

Liveness Analysis and Non-Volatile Promotion. Chinchilla's compiler performs liveness analysis for every variable used in the program to identify protected variables. Variables that are not protected do not need undo logging instrumentation. Chinchilla's liveness analysis calculates the span of code over which a variable may be used without being re-written [3] using a local, context-insensitive, backward CFG traversal from the variable's first use to any definition. Chinchilla leverages LLVM's (conservative) alias analysis: an operation that *may* use a variable starts a live range; only an operation that *must* write to a variable ends its live range. If a variable's live range crosses a checkpoint, the variable is protected and Chinchilla allocates it in non-volatile memory. Chinchilla keeps protected data consistent using undo log.

Undo Logging Instrumentation. Chinchilla's compiler adds undo logging code at accesses to protected variables. Chinchilla inserts a call to `uLog`, which implements undo logging, before every *potential write* to a protected variable that may be the variable's first write since a checkpoint. `uLog` takes the address of the variable as an argument and logs the variable's value before the write executes. Chinchilla uses the log to restore values after a power interruption. The compiler does not instrument accesses to data that do not change throughout the program, such as constant pointers to global arrays. The Chinchilla compiler pre-allocates non-volatile log storage equal in size to the sum of the protected variables' sizes and separated from the protected data store by a



(a) Original tight loop (b) Optimized tight loop

Figure 4: **Tight loop optimization.** (a) Original code and (b) optimized code. In the optimized code, if the system follows the common pass (blue), no undo log (`uLog`) function is called.

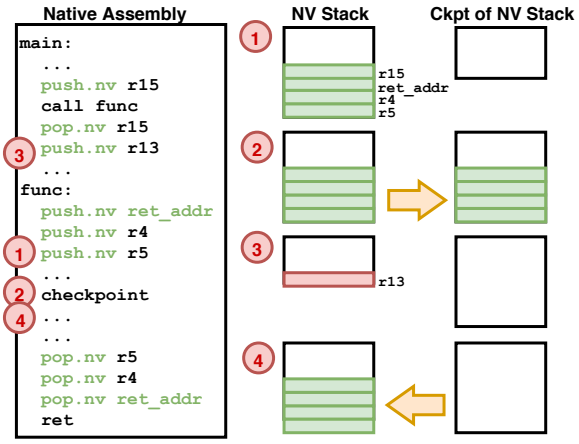


Figure 5: **Non-volatile stack.** The Chinchilla back-end redirects certain `push` and `pop` operation to the non-volatile stack (`push.nv`, `pop.nv`). (1) Some data are pushed to the non-volatile stack. (2) On a checkpoint, only the updated part gets checkpointed. (3) If power fails after non-volatile stack is updated, (4) only the updated part is rolled back.

fixed offset for fast lookups. Chinchilla uses undo logging rather than redo logging (e.g., [35]) because undo logging requires no frequent commit and no instrumentation on read operations. Section 4.3 explains our undo logging implementation.

Memory Layout. Chinchilla organizes protected variables into aligned, fixed-size blocks placed in non-volatile memory with which it associates undo logging metadata; Section 4.3 explains the metadata. Chinchilla uses block metadata, rather than variable- or byte-metadata to amortize its storage overhead. The compiler puts variables smaller than a block in the same block, but disallows a variable to span two blocks. The compiler aligns a variable larger than a block to a block boundary. Chinchilla uses 8 byte blocks, which Section 5 empirically justifies.

Reinitialization. The Chinchilla compiler rewrites the main function to include peripheral reinitialization and log restoration code. The compiler inserts code to restore protected variables from the undo log on reboot. The compiler also inserts a call to a programmer-provided `init` function at the beginning of the main function that reinitializes peripherals on each reboot. Programmers can also perform task-specific re-initialization of the peripherals by setting a non-volatile flag in the task that can be referred in `init`.

Optimized Undo Logging in Tight Loops. Chinchilla's compiler optionally optimizes *tight loops*, which are loops with short bodies that can execute many iterations without exceeding the device's energy buffer. The optimization eliminates per-write undo logging on some variables, instead safely performing undo logging when collecting a checkpoint. Figure 4 shows the optimization applied to the undo logging of `i`. The optimization deletes the undo-logging code in the loop, instead logging (1) in the loop pre-header, and (2) after every checkpoint within the loop (Figure 4b). If the loop checkpoints less than once per iteration (i.e., following Figure 4b's blue path), Chinchilla runs the undo logging function once per checkpoint, rather than once per iteration. If the loop never checkpoints, the undo logging in the pre-header ensures correctness and the optimization never changes program behavior. To avoid optimizing long loops that may checkpoint many times per iteration and lose performance, Chinchilla heuristically selects loops that (1) are inner loops, (2) do not call a function, and (3) have only few (<6) basic blocks in their body.

The optimization idea is different from the static log coalescing from the previous work [4], since it is effectively coalescing the logging function *only when* there is enough energy to run multiple iterations.

4.2 Lightweight Non-Volatile Stack

After compilation, Chinchilla uses a compiler back-end transformation to modify compiled code, to redirect stack accesses that need to be protected but were not visible to Chinchilla's compiler pass, making them refer to Chinchilla's non-volatile stack. These accesses are inserted by the compiler back-end and include saving return addresses, saving and loading caller context, and spilling and reloading registers. Chinchilla must preserve in non-volatile memory the data involved in these accesses when there is a checkpoint between a write and a read of such data.

Chinchilla's back-end replaces these accesses with inlined runtime calls that maintain the non-volatile stack. Chinchilla identifies return address pushes and caller context saves and loads based on the calling convention. Chinchilla identifies register spills and reloads using LLVM IR metadata. We implemented the non-volatile stack transformation in a script that directly modifies assembly; an alternative implementation might modify the LLVM back-end (like Ratchet [51]) at additional implementation effort.

Chinchilla's non-volatile stack has an explicit *top pointer* and a *depth pointer* that tracks the deepest depth to which the top was popped since the most recent checkpoint. Chinchilla uses these pointers to efficiently keep the non-volatile stack consistent across failures. Chinchilla saves with each checkpoint the part of the stack between the top and the depth: the small fraction of the non-volatile stack changed since the last reboot. Clank [25] used a similar differential stack scheme, albeit with architecture support. Figure 5 illustrates the operation of the non-volatile stack management.

4.3 Chinchilla Runtime Library

The Chinchilla runtime library implements adaptive checkpoint collection and restore, undo logging, and non-volatile stack management.

Implementation of `uLog`. Chinchilla's `uLog` function implements undo logging for protected variables. `uLog` takes the address of the variable being accessed as its argument. `uLog` first defensively checks to ensure that it only does undo logging for memory addresses in the range of protected variables, simply returning otherwise. This is necessary because the compiler conservatively inserts undo logging before writes that *may* write to protected variables. Chinchilla compiler omits inserting such defensive check if the accessed data is statically known to be protected.

Chinchilla explicitly tracks whether an access to a variable is its first write since the last checkpoint using an efficient, block-based versioning scheme. Recall that Chinchilla divides memory into blocks of fixed size (Section 4.1). Each block has a one-byte version counter associated with it to track the first write to the block. Chinchilla maintains a global version counter that increments at each collected checkpoint, and at each power interruption. Chinchilla writes the value of the global version counter into a block's version counter each time `uLog` backs up the block (i.e., when a variable contained by the block is written for the first time since a checkpoint.) Chinchilla checks whether a block is in the undo log since the most recent checkpoint by comparing the global version counter to the block's version counter. If a block's version counter is less than the global version counter, the block must be copied to the undo log. Chinchilla clears all blocks' versions when the global version overflows. Figure 6 illustrates how Chinchilla uses versions for undo logging.

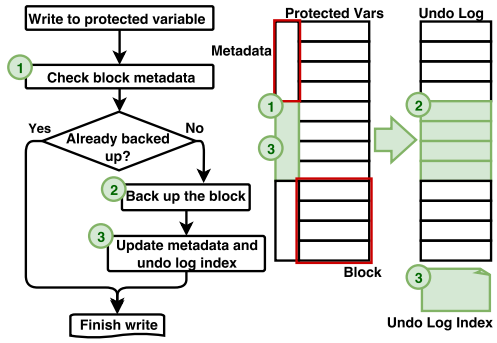


Figure 6: **Undo logging.** Chinchilla (1) checks the metadata and (2) backs up the data before overwriting, (3) updates metadata and the undo log index.

When `uLog` determines that an access is the first write to a variable in a block since the most recent checkpoint, Chinchilla must copy the variable's block to the undo log, which allows restoring the variable's value after a power interruption. A block's undo log location is a fixed offset away from the block; finding the undo log block requires adding the address and offset.

Chinchilla uses an *undo log index* to record backed-up blocks. The index makes it efficient to restore values from the undo log after a power failure, by iterating over the list of index only, not all log storage. Our implementation uses a space-inefficient fixed size index of 1000 entries, although a perfect index with one entry per block of protected variables is also possible. If the fixed-size index were to overflow, it should have the same effect as power failure has; Chinchilla will restart from a checkpoint and subsequently checkpoint more frequently to avoid a second overflow. Chinchilla ensures frequent checkpoints by decreasing its checkpoint timer, which we describe next.

Implementation of Checkpointing. Chinchilla's runtime system implements checkpoint collection and restoration. At a checkpoint, the system backs up the register file, saves the updated part of the non-volatile stack, and clears the undo log index. Chinchilla backs up the register file by saving the contents of registers to a fixed memory region. Chinchilla does not checkpoint the entire non-volatile stack, instead saving the *update* since the last checkpoint, which is contained between the top pointer and depth pointer as discussed in Section 4.2. Chinchilla's checkpoints are double buffered, and if power fails while capturing a checkpoint, Chinchilla reverts to the last successful checkpoint. After a checkpoint, Chinchilla clears its undo log index by resetting the iterator of the index.

Implementation of Restoring. After a power interruption, Chinchilla executes a restore procedure to revert the execution context to the most recent checkpoint before continuing execution. The procedure calls the `init` function, (discussed in Section 4.1) to reconfigure the system's peripherals. It then uses its undo log to revert modified protected variables to their value at the previous checkpoint and restores the changed portion of the non-volatile stack since the previous checkpoint. Finally, the restore routine restores the contents of the register file, restoring the program counter and continuing execution.

If power fails during restoration, Chinchilla continues to try to restore to the same checkpoint after rebooting. Chinchilla keeps its iterator of the undo log in non-volatile memory and during the continuation of the restore procedure Chinchilla can start restoring protected variables from where it left off in the undo log. Assuming the restore procedure successfully reverts at least one protected variable from the undo log, the amount of work in the restore procedure decreases with each attempt. After eventually reverting all entries in the undo log, the only restoration work remaining is to repopulate the register file and continue execution.

Timer Adaptation. Chinchilla uses a *checkpoint timer* to determine when to checkpoint. Chinchilla maintains the timer's interval and configures the timer to count that interval after a reboot. After the timer expires, the next checkpoint call executed collects a checkpoint; before the timer expires, checkpoints do nothing. Chinchilla adjusts the timer's interval during execution to avoid checkpointing too frequently or infrequently. If the interval is too long, power repeatedly fails before the timer expires each time Chinchilla reboots from the last checkpoint. On observing many consecutive failures with no progress, Chinchilla makes a *large decrement* to its checkpoint timer interval by halving its interval. If the checkpoint timer interval is very close to the device's operating period, power may occasionally fail before collecting a checkpoint. On observing a failure before reaching a checkpoint followed by a successfully collected checkpoint (i.e., non-repeated failures), the device makes a *small decrement* to its interval of half the amount of its last change (i.e., small or large).

If the timer interval is very short, Chinchilla may collect checkpoints too frequently. Chinchilla avoids this overhead in two ways. If the checkpoint timer expires twice without a power failure, Chinchilla makes a *large increment* by doubling the timer's interval. Chinchilla increases its checkpoint timer interval using a second *optimization timer* with an interval longer than the checkpoint timer by half the last change in the checkpoint timer's interval. If the optimization timer expires without a power failure and takes a checkpoint, Chinchilla makes a *small increment* to the checkpoint timer interval by assigning the value of the optimization timer interval, and again set the optimization timer interval longer by half the last change in the checkpoint timer's interval.

We implemented both the *checkpoint timer* and the *optimization timer* using a single hardware timer that counts up with two separate handlers. Chinchilla only keeps one *context-insensitive* checkpoint timer interval, assuming the time from boot to power failure is roughly constant regardless of which code is executing. Maintaining different timer intervals may provide a benefit for a system that varies significantly in its operating power (e.g., due to peripheral activity).

4.4 Non-Termination Checker

Our Chinchilla implementation places checkpoints on every basic block, making it simple to statically measure block energy consumption with high precision and ensure the absence of non-termination for a given device's energy buffer. We implemented an energy checker based on CleanCut [13], that measured block energy and compares to device energy automatically.

The checker extracts code between checkpoints from the program's assembly code and generates a measurement binary containing initialization code and the extracted code only. Memory accesses using an unknown reference are redirected to a known location, avoiding referencing invalid memory space as in CleanCut [13]. The checker inserts code that measures energy (i.e., capacitor voltage) at the start and end of the extracted code using EDB [11]. The checker applies this measurement procedure to every basic block in a program and repeatedly executes it multiple times to compensate for measurement noise. If none uses energy that exceeds the device's capacitor energy, the program is unlikely to experience non-termination.

If the checker reveals that a basic block uses more than the capacitor's energy, the block should be subdivided into multiple blocks by the compiler or the programmer. A subdivision is not often necessary: a typical device [47] can run thousands or tens of thousands of instructions before exhausting buffered energy, avoiding non-termination — since our checkpoint blocks are usually small, none of our experiments required subdivision (Section 5.4).

5 Evaluation

We evaluated Chinchilla on *real hardware*, the WISP5 [47] energy harvesting platform equipped with a TI MSP430FR5969 processor. We wirelessly powered the platform using RF energy from the ThingMagic Astra-EX RFID reader at various power levels, placed 20cm apart. We experimented with two WISP hardware configurations: a stock WISP5 (WISP), with the standard 47μF capacitor, and a physically modified WISP5

(WISP-tiny) on which we replaced the standard capacitor with a much smaller 10μF capacitor.

We compared to three previous systems, Alpaca [35], which is the most recent task-based intermittent programming model, Chain [12], another task-based programming model, and Ratchet [51], a compiler-automated approach. Alpaca is a task-based programming and execution model that asks the programmer to write a program as a collection of tasks, complicating programming but leading to high performance by eliminating some inefficiencies of automatic checkpointing systems. Alpaca's tasks have a fixed size and may exceed the device's energy buffer, leading to non-termination and limiting portability. Chain is similar to Alpaca, although with a more complex, channel-based memory model. Ratchet inserts checkpoints automatically while asking nothing of the programmer, but sacrificing performance for this programming simplicity. Ratchet also provides no easy way to check that an inter-checkpoint region will not exceed the device's energy buffer, risking non-termination. Our comparative evaluation shows that Chinchilla's adaptive checkpointing approach is "the best of both worlds," with programmability similar to Ratchet and performance comparable to Alpaca. Moreover, Chinchilla's simple block energy checking procedure allows deploying code with confidence that no block exceeds the device energy buffer, a unique feature that neither Alpaca nor Ratchet provides.

We used the released versions of Alpaca and Chain, directly from the authors. In correspondence with its authors, we ported Ratchet to MSP430 because Ratchet originally targeted ARM only [51]. Our port omits some ARM-specific back-end optimizations from Ratchet, resulting in possibly around 1.6x slowdown on average for our port according to the original work [51].

5.1 Application Benchmarks

We evaluated Chinchilla using all six benchmarks from the Alpaca paper, ported to run on all systems in our setup [35]. The benchmarks are Cold-chain Equipment Monitoring (CEM), Cuckoo Filter (CF), RSA encryption (RSA), Activity Recognition (AR), Bitcount (BC), and Blowfish encryption (BF). For Alpaca, we directly used the benchmarks written by the authors.

CEM reads temperature sensor values and LZW-compresses them. For repeatability, we emulated the sensor with pseudo-random numbers. We used a 512-entry dictionary and a 64-byte compressed block size. CF stores and reads an input data stream using a cuckoo filter with 128 entries. RSA encrypts an eleven character string using RSA with a 64-bit key. AR computes the mean and standard deviation of a window of accelerometer readings to train a nearest neighbor model to detect a shaking movement. We used a window size of three

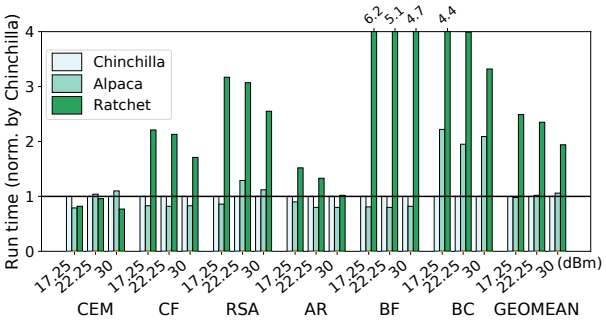


Figure 7: **Run time in different power conditions.**

and read 128 samples from each class (shaking or stationary) in the training phase. BF encrypts a given string of length 13 using blowfish encryption. BC counts the number of one bits in a bitstream. For every result presented, we executed the experiment repeatedly, at most more than 200 times if necessary, until the confidence interval converged into less than 10% of the result.

5.2 Chinchilla is Efficient

Chinchilla ensures a program runs with reasonable overheads in a variety of different wireless power conditions, outperforming state-of-the-art task-based and automatic checkpointing systems in many cases. We measured run time with RF power at 17.25dBm, 22.25dBm, and 30dBm. Figure 7 shows the results.

Chinchilla's run time is faster than the previous programmability-oriented system, Ratchet, in all benchmarks except for CEM, showing an *average speedup around 2.25x*. Even when compared to the performance-oriented Alpaca, Chinchilla shows near-parity performance, with 2% speedup on average. The plot omits data comparing to Chain for brevity; Chinchilla consistently out-performed Chain, with 2.98x average speedup. The main performance benefit of Chinchilla comes from its ability to disable checkpoints, which will be further discussed in Section 5.4.

5.3 Chinchilla is Effectively Portable

Figure 8 shows the energy use of each basic block of Chinchilla in different benchmarks with standard deviation, measured with our checker. We compare block energy to both WISP and the WISP-tiny, shown in the plot as *WISP* and *tiny*. The result from the checker shows that all the benchmarks can run reliably on both platforms, with ample headroom of 2100% (WISP) and 375% (tiny). Thanks to Chinchilla's adaptive checkpointing scheme, this apparent overprovisioning does not impede high performance.

We measured Chinchilla's performance and ability to make progress with different energy buffer sizes (WISP,

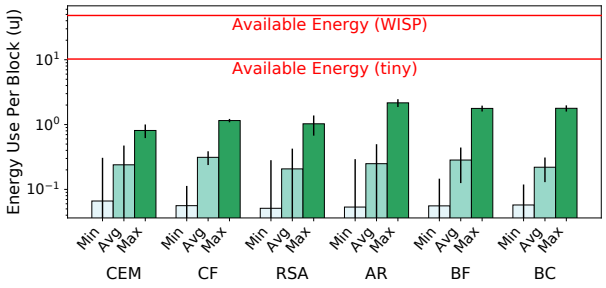


Figure 8: **Energy use of Chinchilla's basic block.** Energy held in WISP (47μF) and tiny (10μF) is also shown.

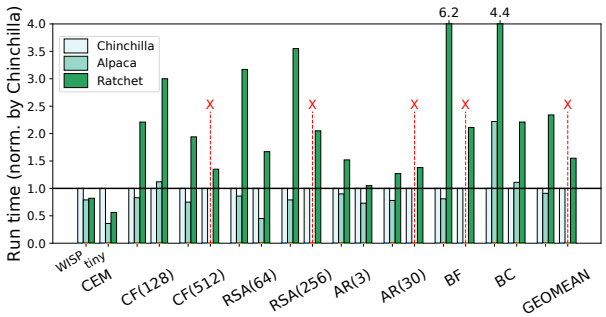


Figure 9: **Execution time in different capacitor size.** Ran on WISP (47μF) and tiny (10μF). A red X indicates the system failed to complete.

tiny), and with varied input sizes. We built variants of CF, RSA, and AR with larger inputs that scale execution time due to an input dependent loop. We increased CF's filter size to 512. We increased RSA's key size to 256. We increased AR's input window size to 30. We performed all tests at 17.25dBm.

The data in Figure 9 show that Chinchilla efficiently operates across a wide range of energy configurations, while Alpaca *fails to complete* in 4 out of 9 cases (CF(512), RSA(256), AR(30), BF) using WISP-tiny.

Ratchet's lack of adaptability makes it slower than Chinchilla across many inputs and energy buffers. Additionally, Ratchet inserts many checkpoints in these applications and never faces non-termination in these data. However, there is no simple way to check that for a different input or hardware configuration Ratchet will avoid non-termination.

5.4 Chinchilla Selectively Checkpoints

We evaluated Chinchilla's ability to adaptively checkpoint only when necessary. We ran complete trials of each application repeatedly and dynamically counted the number of collected checkpoints (Chinchilla, Ratchet) or task transition (Alpaca) and we refer to both as "checkpoint" for brevity.

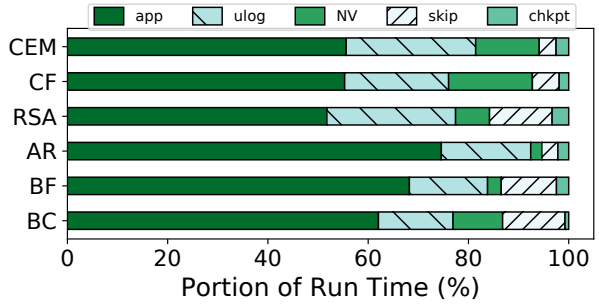


Figure 10: **Overhead breakdown.** Time spent for application code (app), undo log (ulog), non-volatile stack management (NV), skipping disabled checkpoint (skip), and checkpoint and restore (chkpt) is shown.

Table 1 shows the result of the experiment. On average, Alpaca collected 2,185% more checkpoints than Chinchilla and Ratchet collected 21,817% more checkpoints than Chinchilla. The result implies that neither the programmer (Alpaca) nor the compiler (Ratchet) places fixed checkpoints efficiently.

Table 1: **Number of checkpoints taken**

# Chkpt.	CEM	CF	RSA	AR	BF	BC
Chinchilla	30	10	16	26	175	15
Alpaca	1611	452	315	265	1081	710
Ratchet	2319	2478	7643	2911	31881	8907

We characterize the major overheads of Chinchilla in each app to explain its performance. We measured using a Saleae Digital Logic Analyzer timing GPIO pulses instrumented into code to indicate when different operation types occur. To allow timing instrumentation, we measured overhead on continuous power, emulating power failures using a timer. The major overheads are undo logging (ulog), managing the non-volatile stack (NV), skipping disabled checkpoints (skip), and checkpointing and restoring the checkpoint (chkpt).

Figure 10 shows that undo logging is Chinchilla's major overhead. In contrast, checkpointing and restoring is less than 3.5% of run time across benchmarks, illustrating that Chinchilla avoids unnecessary checkpoints. The result shows that Chinchilla can effectively eliminate checkpointing overhead, which is the major source of performance improvement against Ratchet. However, the additional cost of undo-logging and non-volatile stack management for enabling dynamic checkpointing became the new bottleneck of the system.

5.5 Chinchilla Programming is Simple

Chinchilla makes programming simple by allowing the programmer to use all of C, except for dynamic memory allocation, which is uncommon in embedded

code with strict resource constraints. Programming with Chinchilla is simpler than programming with a task-based system. We compare the programmability of Chinchilla against three task-based systems, Alpaca [35], Chain [12], and DINO [31].

There are three aspects of Chinchilla that make it easier to program than task-based models. First, Chinchilla allows using plain C with no special keywords, and Chinchilla's compiler automatically makes code intermittence-safe. Second, Chinchilla allows complex use of pointers, by disambiguating memory references dynamically during undo logging. In comparison, Chain and DINO prohibit pointers to non-volatile memory [12, 31] and Alpaca prohibits some uses of pointers [35]. Third, Chinchilla's block energy checker frees the programmer from reasoning directly about energy consumption while coding. Chinchilla also eliminates the need to rewrite code when hardware or input changes.

Table 2 quantifies programming complexity counting system-specific keywords in our test programs. Chinchilla only requires the programmer to place a checkpoint timer interval tuning function, which our benchmarks do at each outer loop iteration. Chinchilla also allows, but does not require, the programmer to mark atomic regions and none of our applications called for any atomic regions. Compared to other systems, Chinchilla asks very little of the programmer: Alpaca, Chain, and DINO require the programmer to declare system-specific data structures, define tasks, and manually place boundaries and checkpoints.

Table 2: **Summary of programming complexity.**

	App	Chinchilla	Alpaca	Chain	DINO
# Keywords	CEM	1	47	122	13
	CF	1	48	132	11
	RSA	1	67	203	35
	AR	1	45	110	8
	BC	1	49	106	10
	BF	1	42	122	9
Prog. Complexity		Similar to C	High	High	Med
Portability		No Extra Cost	Low	Low	Med
Pointer Support		Always	Med	Low	Low

5.6 Metadata Block Size

As Section 4.1 describes, Chinchilla associates undo logging metadata with a block of data. A larger block size has a lower metadata storage overhead, but incurs a higher run time undo logging cost because the undo log moves data at block granularity. We measured the storage and run time overhead for different block sizes. We omit full data due to space constraints, but we experimentally determined that when the metadata overhead is 12.5% or more (i.e., eight-byte blocks), time overhead is low. Larger blocks had higher run time overhead and we use eight-byte blocks. We also found that few variables (1%–4%) could be kept in SRAM and most were promoted to non-volatile memory because most lifetimes spanned a checkpoint.

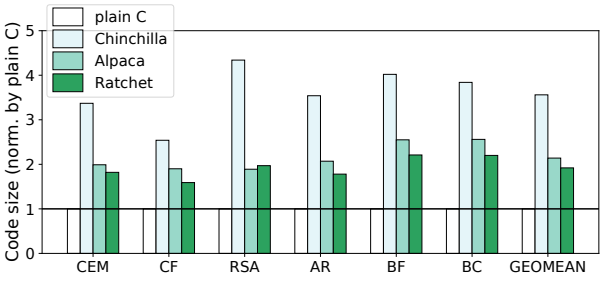


Figure 11: **Compiled code size of different systems.**

5.7 Code Size Increase

Chinchilla inserts checkpointing code between *every* basic block, increasing code size. Figure 11 shows the normalized code size, measured by directly inspecting compiled binaries. Note that while the plain C code is smallest, it does not run correctly on intermittent energy.

All intermittent computing systems see a code size increase due to instrumentation and libraries. Chinchilla has a 3.56x code size increase compared to plain C, 1.66x compared to Alpaca, and 1.86x compared to Ratchet. The code size increase is the cost Chinchilla pays for its performance and reliability benefits. While increased code size may increase instruction cache miss rate, Section 5.2 shows that Chinchilla has higher performance than prior systems regardless of any potential increase.

5.8 Alternate Checkpointing Heuristic

We studied an alternative to Chinchilla's timer-based checkpoint disabling heuristic that decides whether to collect a checkpoint based on whether the checkpoint was used to restore in the recent past. If execution never resumes from a checkpoint, the checkpoint is unlikely to be useful and should be disabled.

We implemented this alternative *history-based* checkpoint disabling heuristic that disables checkpoints that were collected but not used for a fixed period of the execution. The system stores a *score* for each checkpoint that indicates its likely usefulness. On power failure, the system updates a checkpoints' scores, incrementing the score of the checkpoint used for restoration, and decrementing the scores of checkpoints collected and not used. Periodically, the system disables checkpoints with a score below a threshold.

Figure 12 compares the performance of Chinchilla and Chinchilla reimplemented to use this alternative heuristic. We observed that the history-based heuristic was sometimes comparable to Chinchilla's approach, but suffered performance degradation for some bench-

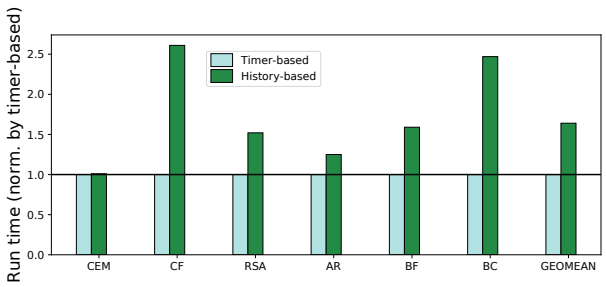


Figure 12: **Run time of two different heuristics.** Run time of timer-based checkpoint disabling versus history-based checkpoint disabling.

marks. The heuristic does especially poorly with functions called from multiple different calling contexts because the score associated with a checkpoint is *context-insensitive*. The timer-based heuristic was more consistent and simpler to implement. We ultimately exclude the history-based heuristic from Chinchilla's design.

6 Related Work

Various prior work influenced the design of Chinchilla. The most related work is on intermittent computation on energy-harvesting devices. Work on maintaining non-volatile memory consistency and approaches that leverage undo logging to maintain consistent execution of the program, such as transactional memory, is related as well. Our work is also related to the prior work that tried to estimate energy use of an arbitrary code.

Intermittent Execution Prior work [36, 44, 51] preserves progress with automatically inserted checkpoints of the execution context. Automatic checkpointing often insert redundant checkpoints, impeding performance. Ensuring progress or atomicity with these techniques is complex because they insert checkpoints arbitrarily. Some systems estimate code energy cost to place checkpoints [4,7,13], but estimating energy in arbitrary code is difficult and error-prone [13]. Task systems ask the programmer to place task boundaries [12, 31, 35], requiring the programmer to form tasks that do not consume too much energy. Mayfly [23] adds real-time constraints on I/O processing to a task system. Forcing the programmer to define tasks complicates programming and offers no simple way to ensure non-termination. Moreover, these models preclude some C features. Chinchilla eliminates programming complexity and allows most of C. Some systems checkpoint "on demand" [5, 6, 26, 45] by monitoring supply voltage. These avoid unnecessary checkpoints, but require extra hardware, and require complex tuning of a checkpoint trigger threshold; a bad threshold risks failing to checkpoint. Non-volatile processors (NVP) [34] change the architecture to save state. Incidental computing [33] and NEOFog [32] optimize the NVP for latency insensitive code and fog computing. Clank [25] implements undo-logging in microarchitecture. Capybara [14] adds a flexible energy storage capacitor, meeting varied energy demand. UFoP [21] assigns a capacitor for each peripheral, and Flicker [22] assists rapid prototyping of an energy-harvesting device. TARDIS and CusTARD [24] keeps time with low power on an energy-harvesting device. Chinchilla requires no architecture or hardware support.

Abundant prior work addressed low-energy embedded systems, but not explicitly intermittent execution. Tock [28] is an OS with multi-tenancy for low-power systems [2]. Dewdrop [8] supports energy-harvesting, but not intermittent execution. Eon [48] allows specifying how tasks of different cost should be scheduled as energy conditions change. ZebraNet [27] used energy-harvesting devices to track wildlife, but with large batteries and solar panels, not intermittent operation. Other work addresses deep neural networks on an energy-harvesting devices [18]. Some work helps develop intermittent code. Wisent [49] and Stork [1] update software on intermittent hardware. Ekho [54] and EDB [11] helps with testing on energy-harvesting devices.

Non-Volatile Memory Consistency Prior work on memory persistency in powered systems support consistency in mixed-volatility memory with access reordering [41, 42, 55]. Others support consistent, non-volatile data structure and file systems [10, 15, 16, 17, 39, 40, 52, 53]. Transactions and transactional memory systems [20, 37, 38, 46] also support consistency and persistence. Chinchilla also supports non-volatile memory consistency (i.e., persistency), but unlike prior work, does so for intermittently powered devices. The rate of failures and constraints on energy and resources faced by Chinchilla makes adopting these solutions difficult.

Energy Measurement CleanCut [13] estimates energy cost of arbitrary code, to aid in checkpoint placement. Other work [4, 7] estimates energy use of code by looking at instruction or cycle count. Both have limitations in precisely estimating the energy use correctly. Chinchilla avoids the problem by confining energy measurement to a basic block. Other works outside the domain of energy-harvesting also tried estimating energy use of code by using evolutionary modeling [29] or by the number of active gates [9]. However, these approaches only estimate the energy use of a processor core, while Chinchilla checker checks the energy of the entire platform.

7 Conclusion

Chinchilla is a fully-automatic, adaptive system that enables correct intermittent execution without additional programming complexity. Automatic compilation and undo logging enables writing unmodified C code. dy-

namic checkpoint adaptation offers portability across platforms, inputs, and environments without recompilation. Chinchilla brings its benefits with low run time cost compared to the state of the art, with an average 2% speedup compared to Alpaca, and a 125% speedup over Ratchet. Chinchilla is the first system to simplify programmability using adaptive checkpoints, and provide strong static assurance of progress without the aid of specialized hardware.

Acknowledgements

Thanks to the anonymous reviewers for the insightful feedback and to Alexei Colin, Emily Ruppel, and Adwait Dongare for the valuable discussion and feedback about the work. We are also grateful to Cristiano Giuffrida for shepherding our final draft. This work is based on work supported by National Science Foundation grant CSR-1526342 and CAREER Award CCF-1751029. Kiwan Maeng was supported by a scholarship from the Korea Foundation for Advanced Studies.

References

[1] AANTJES, H., MAJID, A. Y., PAWEŁCZAK, P., TAN, J., PARKS, A., AND SMITH, J. R. Fast downstream to many (computational) rfids. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE* (2017), IEEE, pp. 1–9.

[2] ADKINS, J., CAMPBELL, B., GHENA, B., JACKSON, N., PANNUTO, P., AND DUTTA, P. The signpost network: Demo abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, ACM, pp. 320–321.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: principles, techniques, and tools*, vol. 2. Addison-wesley Reading, 2007.

[4] BAGHSORKHI, S. S., AND MARGIOLAS, C. Automating efficient variable-grained resiliency for low-power iot systems. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (2018), ACM, pp. 38–49.

[5] BALSAMO, D., WEDDELL, A. S., DAS, A., ARREOLA, A. R., BRUNELLI, D., AL-HASHIMI, B. M., MERRETT, G. V., AND BENINI, L. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35*, 12 (2016), 1968–1980.

[6] BALSAMO, D., WEDDELL, A. S., MERRETT, G. V., AL-HASHIMI, B. M., BRUNELLI, D., AND BENINI, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters 7*, 1 (2015), 15–18.

[7] BHATTI, N. A., AND MOTTOLA, L. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2017), ACM, pp. 209–219.

[8] BUETTNER, M., GREENSTEIN, B., AND WETHERALL, D. Dewdrop: An energy-aware runtime for computational rfid. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 197–210.

[9] CHERUPALLI, H., DUWE, H., YE, W., KUMAR, R., AND SARTORI, J. Determining application-specific peak power and energy requirements for ultra-low power processors. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 3–16.

[10] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[11] COLIN, A., HARVEY, G., LUCIA, B., AND SAMPLE, A. P. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 577–589.

[12] COLIN, A., AND LUCIA, B. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2016), OOPSLA 2016, ACM, pp. 514–530.

[13] COLIN, A., AND LUCIA, B. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction* (2018), ACM, pp. 116–127.

[14] COLIN, A., RUPPEL, E., AND LUCIA, B. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM.

[15] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[16] DOSHI, K., AND VARMAN, P. Wrap: Managing byte-addressable persistent memory. In *Memory Archiecture and Organization Workshop.(MeAOW)* (2012).

[17] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[18] GOBIESKI, G., BECKMANN, N., AND LUCIA, B. Intermittent deep neural network inference.

[19] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* (2001), IEEE, pp. 3–14.

[20] HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[21] HESTER, J., SITANAYAH, L., AND SORBER, J. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2015), SenSys '15, ACM, pp. 5–16.

[22] HESTER, J., AND SORBER, J. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (2017), ACM, p. 19.

[23] HESTER, J., STORER, K., AND SORBER, J. Timely execution on intermittently powered batteryless sensors. In *Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2017), SenSys 2017, ACM.

[24] HESTER, J., TOBIAS, N., RAHMATI, A., SITANAYAH, L., HOLCOMB, D., FU, K., BURLESON, W. P., AND SORBER, J. Persistent clocks for batteryless sensing devices. *ACM Trans. Embed. Comput. Syst. 15*, 4 (Aug. 2016), 77:1–77:28.

[25] HICKS, M. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ACM, pp. 228–240.

[26] JAYAKUMAR, H., RAHA, A., STEVENS, J. R., AND RAGHUNATHAN, V. Energy-aware memory mapping for hybrid framsram mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst. 16*, 3 (Apr. 2017), 65:1–65:23.

[27] JUANG, P., OKI, H., WANG, Y., MARTONOSI, M., PEH, L. S., AND RUBENSTEIN, D. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 96–107.

[28] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 234–251.

[29] LIQAT, U., BANKOVIC, Z., LOPEZ-GARCIA, P., AND HERMENEGILDO, M. V. Inferring energy bounds statically by evolutionary analysis of basic blocks. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)* (2016).

[30] LUCIA, B., BALAJI, V., COLIN, A., MAENG, K., AND RUPPEL, E. Intermittent computing: Challenges and opportunities. In *LIPIcs-Leibniz International Proceedings in Informatics* (2017), vol. 71, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] LUCIA, B., AND RANSFORD, B. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI 2015, ACM, pp. 575–585.

[32] MA, K., LI, X., KANDEMIR, M. T., SAMPSON, J., NARAYANAN, V., LI, J., WU, T., WANG, Z., LIU, Y., AND XIE, Y. Neofog: Nonvolatility-exploiting optimizations for fog computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ACM, pp. 782–796.

[33] MA, K., LI, X., LI, J., LIU, Y., XIE, Y., SAMPSON, J., KANDEMIR, M. T., AND NARAYANAN, V. Incidental computing on iot nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, ACM, pp. 204–218.

[34] MA, K., ZHENG, Y., LI, S., SWAMINATHAN, K., LI, X., LIU, Y., SAMPSON, J., XIE, Y., AND NARAYANAN, V. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 526–537.

[35] MAENG, K., COLIN, A., AND LUCIA, B. Alpaca: Intermittent execution without checkpoints. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2017), OOPSLA 2017, ACM.

[36] MIRHOSEINI, A., SONGHORI, E. M., AND KOUSHANFAR, F. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on* (2013), IEEE, pp. 216–224.

[37] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (1992), 94–162.

[38] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., WOOD, D. A., ET AL. Logtm: log-based transactional memory. In *HPCA* (2006), vol. 6, pp. 254–265.

[39] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), ACM, p. 1.

[40] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 401–410.

[41] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 265–276.

[42] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro 35*, 3 (2015), 125–131.

[43] RANSFORD, B., AND LUCIA, B. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (New York, NY, USA, 2014), MSPC '14, ACM, pp. 5:1–5:3.

[44] RANSFORD, B., SORBER, J., AND FU, K. Mementos: System support for long-running computation on rfid-scale devices. 159–170.

[45] RANSFORD, B., SORBER, J., AND FU, K. Mementos: System support for long-running computation on rfid-scale devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 159–170.

[46] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 187–197.

[47] SAMPLE, A. P., YEAGER, D. J., POWLEDGE, P. S., MAMISHEV, A. V., AND SMITH, J. R. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement 57*, 11 (2008), 2608–2615.

[48] SORBER, J., KOSTADINOV, A., GARBER, M., BRENNAN, M., CORNER, M. D., AND BERGER, E. D. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2007), SenSys '07, ACM, pp. 161–174.

[49] TAN, J., PAWEŁCZAK, P., PARKS, A., AND SMITH, J. R. Wisent: Robust downstream communication and storage for computational rfids. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE* (2016), IEEE, pp. 1–9.

[50] TI INC. Products for msp430frxx fram. http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/ultra-low-power/msp430frxx-fram/products.page, 2017. Accessed: 2017-04-08.

[51] VAN DER WOUDE, J., AND HICKS, M. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 17–32.

[52] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 5–5.

[53] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 91–104.

[54] ZHANG, H., SALAJEGHEH, M., FU, K., AND SORBER, J. Ekho: Bridging the gap between simulation and reality in tiny energy-harvesting sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems* (New York, NY, USA, 2011), HotPower '11, ACM, pp. 9:1–9:5.

[55] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 421–432.

Arachne: Core-Aware Thread Management

Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout
{hq6,qianli,jspeiser,kraftp,ouster}@cs.stanford.edu
Stanford University

Abstract

Arachne is a new user-level implementation of threads that provides both low latency and high throughput for applications with extremely short-lived threads (only a few microseconds). Arachne is *core-aware*: each application determines how many cores it needs, based on its load; it always knows exactly which cores it has been allocated, and it controls the placement of its threads on those cores. A central core arbiter allocates cores between applications. Adding Arachne to memcached improved SLO-compliant throughput by 37%, reduced tail latency by more than 10x, and allowed memcached to coexist with background applications with almost no performance impact. Adding Arachne to the RAMCloud storage system increased its write throughput by more than 2.5x. The Arachne threading library is optimized to minimize cache misses; it can initiate a new user thread on a different core (with load balancing) in 320 ns. Arachne is implemented entirely at user level on Linux; no kernel modifications are needed.

1 Introduction

Advances in networking and storage technologies have made it possible for datacenter services to operate at exceptionally low latencies [5]. As a result, a variety of low-latency services have been developed in recent years, including FaRM [11], Memcached [23], MICA [20], RAMCloud [30], and Redis [34]. They offer end-to-end response times as low as 5 µs for clients within the same datacenter and they have internal request service times as low as 1–2 µs. These systems employ a variety of new techniques to achieve their low latency, including polling instead of interrupts, kernel bypass, and run to completion [6, 31].

However, it is difficult to construct services that provide both low latency and high throughput. Techniques for achieving low latency, such as reserving cores for peak throughput or using polling instead of interrupts, waste resources. Multi-level services, in which servicing one request may require nested requests to other servers (such as for replication), create additional opportunities for resource underutilization, particularly if they use polling to reduce latency. Background activities within a service, such as garbage collection, either require additional reserved (and hence underutilized) resources, or risk interference with foreground request servicing. Ideally, it should be possible to colocate throughput-oriented services such as MapReduce [10] or video processing [22] with low-latency services, such that resources are fully occupied by the throughput-oriented services when not needed by the low-latency services. However, this is rarely attempted in practice because it impacts the performance of the latency-sensitive services.

One of the reasons it is difficult to combine low latency and high throughput is that applications must manage their parallelism with a virtual resource (threads); they cannot tell the operating system how many physical resources (cores) they need, and they do not know which cores have been allocated for their use. As a result, applications cannot adjust their internal parallelism to match the resources available to them, and they cannot use application-specific knowledge to optimize their use of resources. This can lead to both under-utilization and over-commitment of cores, which results in poor resource utilization and/or suboptimal performance. The only recourse for applications is to pin threads to cores; this results in underutilization of cores within the application and does not prevent other applications from being scheduled onto the same cores.

Arachne is a thread management system that solves these problems by giving applications visibility into the physical resources they are using. We call this approach *core-aware thread management*. In Arachne, application threads are managed entirely at user level; they are not visible to the operating system. Applications negotiate with the system over cores, not threads. Cores are allocated for the exclusive use of individual applications and remain allocated to an application for long intervals (tens of milliseconds). Each application always knows exactly which cores it has been allocated and it decides how to schedule application threads on cores. A *core arbiter* decides how many cores to allocate to each application, and adjusts the allocations in response to changing application requirements.

User-level thread management systems have been implemented many times in the past [39, 14, 4] and the basic features of Arachne were prototyped in the early 1990s in the form of scheduler activations [2]. Arachne is novel in the following ways:

- Arachne contains mechanisms to estimate the number of cores needed by an application as it runs.
- Arachne allows each application to define a *core policy*, which determines at runtime how many cores the application needs and how threads are placed on the available cores.
- The Arachne runtime was designed to minimize cache misses. It uses a novel representation of scheduling

information with no ready queues, which enables low-latency and scalable mechanisms for thread creation, scheduling, and synchronization.
- Arachne provides a simpler formulation than scheduler activations, based on the use of one kernel thread per core.
- Arachne runs entirely outside the kernel and needs no kernel modifications; the core arbiter is implemented at user level using the Linux cpuset mechanism. Arachne applications can coexist with traditional applications that do not use Arachne.

We have implemented the Arachne runtime and core arbiter in C++ and evaluated them using both synthetic benchmarks and the memcached and RAMCloud storage systems. Arachne can initiate a new thread on a different core (with load balancing) in about 320 ns, and an application can obtain an additional core from the core arbiter in 20–30 µs. When Arachne was added to memcached, it reduced tail latency by more than 10x and allowed 37% higher throughput at low latency. Arachne also improved performance isolation; a background video processing application could be colocated with memcached with almost no impact on memcached's latency. When Arachne was added to the RAMCloud storage system, it improved write throughput by more than 2.5x.

2 The Threading Problem

Arachne was motivated by the challenges in creating services that process very large numbers of very small requests. These services can be optimized for low latency or for high throughput, but it is difficult to achieve both with traditional threads implemented by the operating system.

As an example, consider memcached [23], a widely used in-memory key-value-store. Memcached processes requests in about 10 µs. Kernel threads are too expensive to create a new one for each incoming request, so memcached uses a fixed-size pool of worker threads. New connections are assigned statically to worker threads in a round-robin fashion by a separate dispatch thread.

The number of worker threads is fixed when memcached starts, which results in several inefficiencies. If the number of cores available to memcached is smaller than the number of workers, the operating system will multiplex workers on a single core, resulting in long delays for requests sent to descheduled workers. For best performance, one core must be reserved for each worker thread. If background tasks are run on the machine during periods of low load, they are likely to interfere with the memcached workers, due to the large number of distinct worker threads. Furthermore, during periods of low load, each worker thread will be lightly loaded, increasing the risk that its core will enter power-saving states with high-latency wakeups. Memcached would perform better if it could scale back during periods of low load to use a smaller number of kernel threads (and cores) more intensively.

In addition, memcached's static assignment of connections to workers can result in load imbalances under skewed workloads, with some worker threads overloaded and others idle. This can impact both latency and throughput.

The RAMCloud storage system provides another example [30]. RAMCloud is an in-memory key-value store that processes small read requests in about 2 µs. Like memcached, it is based on kernel threads. A dispatch thread handles all network communication and polls the NIC for incoming packets using kernel bypass. When a request arrives, the dispatch thread delegates it to one of a collection of worker threads for execution; this approach avoids problems with skewed workloads. The number of active worker threads varies based on load. The maximum number of workers is determined at startup, which creates issues similar to memcached.

RAMCloud implements nested requests, which result in poor resource utilization because of microsecond-scale idle periods that cannot be used. When a worker thread receives a write request, it sends copies of the new value to backup servers and waits for those requests to return before responding to the original request. All of the replication requests complete within 7-8 µs, so the worker busy-waits for them. If the worker were to sleep, it would take several microseconds to wake it up again; in addition, context-switching overheads are too high to get much useful work done in such a short time. As a result, the worker thread's core is wasted for 70-80% of the time to process a write request; write throughput for a server is only about 150 kops/sec for small writes, compared with about 1 Mops/sec for small reads.

The goal for Arachne is to provide a thread management system that allows a better combination of low latency and high throughput. For example, each application should match its workload to available cores, taking only as many cores as needed and dynamically adjusting its internal parallelism to reflect the number of cores allocated to it. In addition, Arachne should provide an implementation of user-level threads that is efficient enough to be used for very short-lived threads, and that allows useful work to be done during brief blockages such as those for nested requests.

Although some existing applications will benefit from Arachne, we expect Arachne to be used primarily for new *granular applications* whose threads have lifetimes of only a few microseconds. These applications are difficult or impossible to build today because of inefficiencies in current threading infrastructure.

3 Arachne Overview

Figure 1 shows the overall architecture of Arachne. Three components work together to implement Arachne threads. The *core arbiter* consists of a stand-alone user process plus

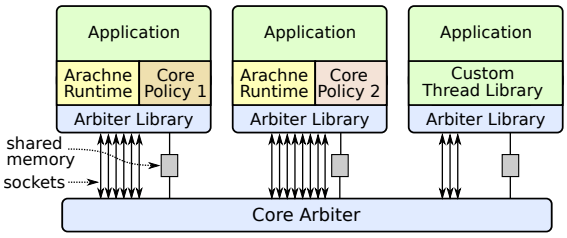


Figure 1: The Arachne architecture. The core arbiter communicates with each application using one socket for each kernel thread in the application, plus one page of shared memory.

a small library linked into each application. The *Arachne runtime* and *core policies* are libraries linked into applications. Different applications can use different core policies. An application can also substitute its own threading library for the Arachne runtime and core policy, while still using the core arbiter.

The core arbiter is a user-level process that manages cores and allocates them to applications. It collects information from each application about how many cores it needs and uses a simple priority mechanism to divide the available cores among competing applications. The core arbiter adjusts the core allocations as application requirements change. Section 4 describes the core arbiter in detail.

The Arachne runtime creates several kernel threads and uses them to implement user threads, which are used by Arachne applications. The Arachne user thread abstraction contains facilities similar to thread packages based on kernel threads, including thread creation and deletion, locks, and condition variables. However, all operations on user threads are carried out entirely at user level without making kernel calls, so they are an order of magnitude faster than operations on kernel threads. Section 5 describes the implementation of the Arachne runtime in more detail.

The Arachne runtime works together with a core policy, which determines how cores are used by that application. The core policy computes the application's core requirements, using performance information gathered by the Arachne runtime. It also determines which user threads run on which cores. Each application chooses its core policy. Core policies are discussed in Section 6.

Arachne uses kernel threads as a proxy for cores. Each kernel thread created by the runtime executes on a separate core and has exclusive access to that core while it is running. When the arbiter allocates a core to an application, it unblocks one of the application's kernel threads on that core; when the core is removed from an application, the kernel thread running on that core blocks. The Arachne runtime runs a simple dispatcher in each kernel thread, which multiplexes several user threads on the associated core.

Arachne uses a cooperative multithreading model for user threads: the runtime does not preempt a user thread once it has begun executing. If a user thread needs to execute for a long time without blocking, it must occasionally invoke a `yield` method, which allows other threads to run before the calling thread continues. We expect most threads to either block or complete quickly, so it should rarely be necessary to invoke `yield`.

One potential problem with a user-level implementation of threads is that a user thread might cause the underlying kernel thread to block. This could happen, for example, if the user thread invokes a blocking kernel call or incurs a page fault. This prevents the kernel thread from running other user threads until the kernel call or page fault completes. Previous implementations of user-level threads have attempted to work around this inefficiency in a variety of ways, often involving complex kernel modifications.

Arachne does not take any special steps to handle blocking kernel calls or page faults. Most modern operating systems support asynchronous I/O, so I/O can be implemented without blocking the kernel thread. Paging is almost never cost-effective today, given the low cost of memory and the large sizes of memories. Modern servers rarely incur page faults except for initial loading, such as when an application starts or a file is mapped into virtual memory. Thus, for simplicity, Arachne does not attempt to make use of the time when a kernel thread is blocked for a page fault or kernel call.

Note: we use the term *core* to refer to any hardware mechanism that can support an independent thread of computation. In processors with hyperthreading, we think of each hyperthread as a separate logical core, even though some of them share a single physical core.

4 The Core Arbiter

This section describes how the core arbiter claims control over (most of) the system's cores and allocates them among applications. The core arbiter has three interesting features. First, it implements core management entirely at user level using existing Linux mechanisms; it does not require any kernel changes. Second, it coexists with existing applications that don't use Arachne. And third, it takes a cooperative approach to core management, both in its priority mechanism and in the way it preempts cores from applications.

The core arbiter runs as a user process with root privilege and uses the Linux *cpuset* mechanism to manage cores. A cpuset is a collection of one or more cores and one or more banks of memory. At any given time, each kernel thread is assigned to exactly one cpuset, and the Linux scheduler ensures that the thread executes only on cores in that cpuset. By default, all threads run in a cpuset containing all cores and all memory banks. The core arbiter uses cpusets to allocate specific cores to specific applications.

The core arbiter divides cores into two groups: *man-*

aged cores and *unmanaged cores*. Managed cores are allocated by the core arbiter; only the kernel threads created by Arachne run on these cores. Unmanaged cores continue to be scheduled by Linux. They are used by processes that do not use Arachne, and also by the core arbiter itself. In addition, if an Arachne application creates new kernel threads outside Arachne, for example, using `std::thread`, these threads will run on the unmanaged cores.

When the core arbiter starts up, it creates one cpuset for unmanaged cores (the *unmanaged cpuset*) and places all of the system's cores into that set. It then assigns every existing kernel thread (including itself) to the unmanaged cpuset; any new threads spawned by these threads will also run on this cpuset. The core arbiter also creates one *managed cpuset* corresponding to each core, which contains that single core but initially has no threads assigned to it. To allocate a core to an Arachne application, the arbiter removes that core from the unmanaged cpuset and assigns an Arachne kernel thread to the managed cpuset for that core. When a core is no longer needed by any Arachne application, the core arbiter adds the core back to the unmanaged cpuset.

This scheme allows Arachne applications to coexist with traditional applications whose threads are managed by the Linux kernel. Arachne applications receive preferential access to cores, except that the core arbiter reserves at least one core for the unmanaged cpuset.

The Arachne runtime communicates with the core arbiter using three methods in the arbiter's library package:

- `setRequestedCores`: invoked by the runtime whenever its core needs change; indicates the total number of cores needed by the application at various priority levels (see below for details).
- `blockUntilCoreAvailable`: invoked by a kernel thread to identify itself to the core arbiter and put the kernel thread to sleep until it is assigned a core. At that point the kernel thread wakes up and this method returns the identifier of the assigned core.
- `mustReleaseCore`: invoked periodically by the runtime; a true return value means that the calling kernel thread should invoke `blockUntilCoreAvailable` to return its core to the arbiter.

Normally, the Arachne runtime handles all communication with the core arbiter, so these methods are invisible to applications. However, an application can implement its own thread and core management by calling the arbiter library package directly.

The methods described above communicate with the core arbiter using a collection of Unix domain sockets and a shared memory page (see Figure 1). The arbiter library opens one socket for each kernel thread. This socket is used to send requests to the core arbiter, and it is also used to put the kernel thread to sleep when it has no assigned core. The shared memory page is used by the core arbiter to pass information to the arbiter library; it is written by the core arbiter and is read-only to the arbiter library.

When the Arachne runtime starts up, it invokes `setRequestedCores` to specify the application's initial core requirements; `setRequestedCores` sends a message to the core arbiter over a socket. Then the runtime creates one kernel thread for each core on the machine; all of these threads invoke `blockUntilCoreAvailable`. `blockUntilCoreAvailable` sends a request to the core arbiter over the socket belonging to that kernel thread and then attempts to read a response from the socket. This has two effects: first, it notifies the core arbiter that the kernel thread is available for it to manage (the request includes the Linux identifier for the thread); second, the socket read puts the kernel thread to sleep.

At this point the core arbiter knows about the application's core requirements and all of its kernel threads, and the kernel threads are all blocked. When the core arbiter decides to allocate a core to the application, it chooses one of the application's blocked kernel threads to run on that core. It assigns that thread to the cpuset corresponding to the allocated core and then sends a response message back over the thread's socket. This causes the thread to wake up, and Linux will schedule the thread on the given core; the `blockUntilCoreAvailable` method returns, with the core identifier as its return value. The kernel thread then invokes the Arachne dispatcher to run user threads.

If the core arbiter wishes to reclaim a core from an application, it asks the application to release the core. The core arbiter does not unilaterally preempt cores, since the core's kernel thread might be in an inconvenient state (e.g. it might have acquired an important spin lock); abruptly stopping it could have significant performance consequences for the application. So, the core arbiter sets a variable in the shared memory page, indicating which core(s) should be released. Then it waits for the application to respond.

Each kernel thread is responsible for testing the information in shared memory at regular intervals by invoking `mustReleaseCore`. The Arachne runtime does this in its dispatcher. If `mustReleaseCore` returns true, then the kernel thread cleans up as described in Section 5.4 and invokes `blockUntilCoreAvailable`. This notifies the core arbiter and puts the kernel thread to sleep. At this point, the core arbiter can reallocate the core to a different application.

The communication mechanism between the core arbiter and applications is intentionally asymmetric: requests from applications to the core arbiter use sockets, while requests from the core arbiter to applications use shared memory. The sockets are convenient because they allow the core arbiter to sleep while waiting for requests; they also allow application kernel threads to sleep while waiting for cores to be assigned. Socket communication is

relatively expensive (several microseconds in each direction), but it only occurs when application core requirements change, which we expect to be infrequent. The shared memory page is convenient because it allows the Arachne runtime to test efficiently for incoming requests from the core arbiter; these tests are made frequently (every pass through the user thread dispatcher), so it is important that they are fast and do not involve kernel calls.

Applications can delay releasing cores for a short time in order to reach a convenient stopping point, such as a time when no locks are held. The Arachne runtime will not release a core until the dispatcher is invoked on that core, which happens when a user thread blocks, yields, or exits.

If an application fails to release a core within a timeout period (currently 10 ms), then the core arbiter will forcibly reclaim the core. It does this by reassigning the core's kernel thread to the unmanaged cpuset. The kernel thread will be able to continue executing, but it will probably experience degraded performance due to interference from other threads in the unmanaged cpuset.

The core arbiter uses a simple priority mechanism for allocating cores to applications. Arachne applications can request cores on different priority levels (the current implementation supports eight). The core arbiter allocates cores from highest priority to lowest, so low-priority applications may receive no cores. If there are not enough cores for all of the requests at a particular level, the core arbiter divides the cores evenly among the requesting applications. The core arbiter repeats this computation whenever application requests change. The arbiter allocates all of the hyperthreads of a particular hardware core to the same application whenever possible. The core arbiter also attempts to keep all of an application's cores on the same socket.

This policy for core allocation assumes that applications (and their users) will cooperate in their choice of priority levels: a misbehaving application could starve other applications by requesting all of its cores at the highest priority level. Anti-social behavior could be prevented by requiring applications to authenticate with the core arbiter when they first connect, and allowing system administrators to set limits for each application or user. We leave such a mechanism to future work.

5 The Arachne Runtime

This section discusses how the Arachne runtime implements user threads. The most important goal for the runtime is to provide a fast and scalable implementation of user threads for modern multi-core hardware. We want Arachne to support granular computations, which consist of large numbers of extremely short-lived threads. For example, a low latency server might create a new thread for each incoming request, and the request might take only a microsecond or two to process; the server might process millions of these requests per second.

5.1 Cache-optimized design

The performance of the Arachne runtime is dominated by cache misses. Most threading operations, such as creating a thread, acquiring a lock, or waking a blocked thread, are relatively simple, but they involve communication between cores. Cross-core communication requires cache misses. For example, to transfer a value from one core to another, it must be written on the source core and read on the destination core. This takes about three cache miss times: the write will probably incur a cache miss to first read the data; the write will then invalidate the copy of the data in the destination cache, which takes about the same time as a cache miss; finally, the read will incur a cache miss to fetch the new value of the data. Cache misses can take from 50-200 cycles, so even if an operation requires only a single cache miss, the miss is likely to cost more than all of the other computation for the operation. On our servers, the cache misses to transfer a value from one core to another in the same socket take 7-8x as long as a context switch between user threads on the same core. Transfers between sockets are even more expensive. Thus, our most important goal in implementing user threads was to minimize cache misses.

The effective cost of a cache miss can be reduced by performing other operations concurrently with the miss. For example, if several cache misses occur within a few instructions of each other, they can all be completed for the cost of a single miss (modern processors have out-of-order execution engines that can continue executing instructions while waiting for cache misses, and each core has multiple memory channels). Thus, additional cache misses are essentially free. However, modern processors have an out-of-order execution limit of about 100 instructions, so code must be designed to concentrate likely cache misses near each other.

Similarly, a computation that takes tens of nanoseconds in isolation may actually have zero marginal cost if it occurs in the vicinity of a cache miss; it will simply fill the time while the cache miss is being processed. Section 5.3 will show how the Arachne dispatcher uses this technique to hide the cost of seemingly expensive code.

5.2 Thread creation

Many user-level thread packages, such as the one in Go [14], create new threads on the same core as the parent; they use work stealing to balance load across cores. This avoids cache misses at thread creation time. However, work stealing is an expensive operation (it requires cache misses), which is particularly noticeable for short-lived threads. Work stealing also introduces a time lag before a thread is stolen to an unloaded core, which impacts service latency. For Arachne we decided to perform load-balancing at thread creation time; our goal is to get a new

thread on an unloaded core as quickly as possible. By optimizing this mechanism based on cache misses, we were able to achieve thread creation times competitive with systems that create child threads on the parent's core.

Cache misses can occur during thread creation for the following reasons:

- **Load balancing**: Arachne must choose a core for the new thread in a way that balances load across available cores; cache misses are likely to occur while fetching shared state describing current loads.
- **State transfer**: the address and arguments for the thread's top-level method must be transferred from the parent's core to the child's core.
- **Scheduling**: the parent must indicate to the child's core that the child thread is runnable.
- **Thread context**: the context for a thread consists of its call stack, plus metadata used by the Arachne runtime, such as scheduling state and saved execution state when the thread is not running. Depending on how this information is managed, it can result in additional cache misses.

We describe below how Arachne can create a new user thread in four cache miss times.

In order to minimize cache misses for thread contexts, Arachne binds each thread context to a single core (the context is only used by a single kernel thread). Each user thread is assigned to a thread context when it is created, and the thread executes only on the context's associated core. Most threads live their entire life on a single core. A thread moves to a different core only as part of an explicit migration. This happens only in rare situations such as when the core arbiter reclaims a core. A thread context remains bound to its core after its thread completes, and Arachne reuses recently-used contexts when creating new threads. If threads have short lifetimes, it is likely that the context for a new thread will already be cached.

To create a new user thread, the Arachne runtime must choose a core for the thread and allocate one of the thread contexts associated with that core. Each of these operations will probably result in cache misses, since they manipulate shared state. In order to minimize cache misses, Arachne uses the same shared state to perform both operations simultaneously. The state consists of a 64-bit `maskAndCount` value for each active core. 56 bits of the value are a bit mask indicating which of the core's thread contexts are currently in use, and the remaining 8 bits are a count of the number of ones in the mask.

When creating new threads, Arachne uses the "power of two choices" approach for load balancing [26]. It selects two cores at random, reads their `maskAndCount` values, and selects the core with the fewest active thread contexts. This will likely result in a cache miss for each `maskAndCount`, but they will be handled concurrently so the total delay is that of a single miss. Arachne then scans the mask bits for the chosen core to find an available thread context and uses an atomic compare-and-swap operation to update the `maskAndCount` for the chosen core. If the compare-and-swap fails because of a concurrent update, Arachne rereads the `maskAndCount` for the chosen core and repeats the process of allocating a thread context. This creation mechanism is scalable: with a large number of cores, multiple threads can be created simultaneously on different cores.

Once a thread context has been allocated, Arachne copies the address and arguments for the thread's top-level method into the context and schedules the thread for execution by setting a word-sized variable in the context to indicate that the thread is runnable. In order to minimize cache misses, Arachne uses a single cache line to hold all of this information. This limits argument lists to 6 one-word parameters on machines with 64-byte cache lines; larger parameter lists must be passed by reference, which will result in additional cache misses.

With this mechanism, a new thread can be invoked on a different core in four cache miss times. One cache miss is required to read the `maskAndCount` and three cache miss times are required to transfer the line containing the method address and arguments and the scheduling flag, as described in Section 5.1.

The `maskAndCount` variable limits Arachne to 56 threads per core at a given time. As a result, programming models that rely on large numbers of blocked threads may be unsuitable for use with Arachne.

5.3 Thread scheduling

The traditional approach to thread scheduling uses one or more ready queues to identify runnable threads (typically one queue per core, to reduce contention), plus a scheduling state variable for each thread, which indicates whether that thread is runnable or blocked. This representation is problematic from the standpoint of cache misses. Adding or removing an entry to/from a ready queue requires updates to multiple variables. Even if the queue is lockless, this is likely to result in multiple cache misses when the queue is shared across cores. Furthermore, we expect sharing to be common: a thread must be added to the ready queue for its core when it is awakened, but the wakeup typically comes from a thread on a different core.

In addition, the scheduling state variable is subject to races. For example, if a thread blocks on a condition variable, but another thread notifies the condition variable before the blocking thread has gone to sleep, a race over the scheduling state variable could cause the wakeup to be lost. This race is typically eliminated with a lock that controls access to the state variable. However, the lock results in additional cache misses, since it is shared across cores.

In order to minimize cache misses, Arachne does not use ready queues. Instead of checking a ready queue, the Arachne dispatcher repeatedly scans all of the active user

thread contexts associated with the current core until it finds one that is runnable. This approach turns out to be relatively efficient, for two reasons. First, we expect only a few thread contexts to be occupied for a core at a given time (there is no need to keep around blocked threads for intermittent tasks; a new thread can be created for each task). Second, the cost of scanning the active thread contexts is largely hidden by an unavoidable cache miss on the scheduling state variable for the thread that woke up. This variable is typically modified by a different core to wake up the thread, which means the dispatcher will have to take a cache miss to observe the new value. 100 or more cycles elapse between when the previous value of the variable is invalidated in the dispatcher's cache and the new value can be fetched; a large number of thread contexts can be scanned during this time. Section 7.4 evaluates the cost of this approach.

Arachne also uses a new lockless mechanism for scheduling state. The scheduling state of a thread is represented with a 64-bit `wakeupTime` variable in its thread context. The dispatcher considers a thread runnable if its `wakeupTime` is less than or equal to the processor's fine-grain cycle counter. Before transferring control to a thread, the dispatcher sets its `wakeupTime` to the largest possible value. `wakeupTime` doesn't need to be modified when the thread blocks: the large value will prevent the thread from running again. To wake up the thread, `wakeupTime` is set to 0. This approach eliminates the race condition described previously, since `wakeupTime` is not modified when the thread blocks; thus, no synchronization is needed for access to the variable.

The `wakeupTime` variable also supports timer-based wakeups. If a thread wishes to sleep for a given time period, or if it wishes to add a timeout to some other blocking operation such as a condition `wait`, it can set `wakeupTime` to the desired wakeup time before blocking. A single test in the Arachne dispatcher detects both normal unblocks and timer-based unblocks.

Arachne exports the `wakeupTime` mechanism to applications with two methods:

- `block(time)` will block the current user thread. The `time` argument is optional; if it is specified, `wakeupTime` is set to this value (using compare-and-swap to detect concurrent wakeups).
- `signal(thread)` will set the given user thread's `wakeupTime` to 0.

These methods make it easy to construct higher-level synchronization and scheduling operators. For example, the `yield` method, which is used in cooperative multithreading to allow other user threads to run, simply invokes `block(0)`.

5.4 Adding and releasing cores

When the core arbiter allocates a new core to an application, it wakes up one of the kernel threads that was blocked in `blockUntilCoreAvailable`. The kernel thread notifies the core policy of the new core as described in Section 6 below, then it enters the Arachne dispatcher loop.

When the core arbiter decides to reclaim a core from an application, `mustReleaseCore` will return true in the Arachne dispatcher running on the core. The kernel thread modifies its `maskAndCount` to prevent any new threads from being placed on it, then it notifies the core policy of the reclamation. If any user threads exist on the core, the Arachne runtime migrates them to other cores. To migrate a thread, Arachne selects a new core (destination core) and reserves an unoccupied thread context on that core using the same mechanism as for thread creation. Arachne then exchanges the context of the thread being migrated with the unoccupied context, so that the thread's context is rebound to the destination core and the unused context from the destination core is rebound to the source core. Once all threads have been migrated away, the kernel thread on the reclaimed core invokes `blockUntilCoreAvailable`. This notifies the core arbiter that the core is no longer in use and puts the kernel thread to sleep.

6 Core Policies

One of our goals for Arachne is to give applications precise control over their usage of cores. For example, in RAMCloud the central dispatch thread is usually the performance bottleneck. Thus, it makes sense for the dispatch thread to have exclusive use of a core. Furthermore, the other hyperthread on the same physical core should be idle (if both hyperthreads are used simultaneously, they each run about 30% slower than if only one hyperthread is in use). In other applications it might be desirable to colocate particular threads on hyperthreads of the same core or socket, or to force all low-priority background threads to execute on a single core in order to maximize the resources available for foreground request processing.

The Arachne runtime does not implement the policies for core usage. These are provided in a separate *core policy* module. Each application selects a particular core policy at startup. Writing high-performance core policies is likely to be challenging, particularly for policies that deal with NUMA issues and hyperthreads. We hope that a small collection of reusable policies can be developed to meet the needs of most applications, so that it will rarely be necessary for an application developer to implement a custom core policy.

In order to manage core usage, the core policy must know which cores have been assigned to the application. The Arachne runtime provides this information by invoking a method in the core policy whenever the application gains or loses cores.

When an application creates a new user thread, it specifies an integer *thread class* for the thread. Thread classes are used by core policies to manage user threads; each

thread class corresponds to a particular level of service, such as "foreground thread" or "background thread." Each core policy defines its own set of valid thread classes. The Arachne runtime stores thread classes with threads, but has no knowledge of how they are used.

The core policy uses thread classes to manage the placement of new threads. When a new thread is created, Arachne invokes a method `getCores` in the core policy, passing it the thread's class. The `getCores` method uses the thread class to select one or more cores that are acceptable for the thread. The Arachne runtime places the new thread on one of those cores using the "power of two choices" mechanism described in Section 5. If the core policy wishes to place the new thread on a specific core, `getCores` can return a list with a single entry. Arachne also invokes `getCores` to find a new home for a thread when it must be migrated as part of releasing a core.

One of the unusual features of Arachne is that each application is responsible for determining how many cores it needs; we call this *core estimation*, and it is handled by the core policy. The Arachne runtime measures two statistics for each core, which it makes available to the core policy for its use in core estimation. The first statistic is *utilization*, which is the average fraction of time that each Arachne kernel thread spends executing user threads. The second statistic is *load factor*, which is an estimate of the average number of runnable user threads on that core. Both of these are computed with a few simple operations in the Arachne dispatching loop.

6.1 Default core policy

Arachne currently includes one core policy; we used the default policy for all of the performance measurements in Section 7. The default policy supports two thread classes: exclusive and normal. Each exclusive thread runs on a separate core reserved for that particular thread; when an exclusive thread is blocked, its core is idle. Exclusive threads are useful for long-running dispatch threads that poll. Normal threads share a pool of cores that is disjoint from the cores used for exclusive threads; there can be multiple normal threads assigned to a core at the same time.

6.2 Core estimation

The default core policy requests one core for each exclusive thread, plus additional cores for normal threads. Estimating the cores required for the normal threads requires making a tradeoff between core utilization and fast response time. If we attempt to keep cores busy 100% of the time, fluctuations in load will create a backlog of pending threads, resulting in delays for new threads. On the other hand, we could optimize for fast response time, but this would result in low utilization of cores. The more bursty a workload, the more resources it must waste in order to get fast response.

The default policy uses different mechanisms for scaling up and scaling down. The decision to scale up is based on load factor: when the average load factor across all cores running normal threads reaches a threshold value, the core policy increases its requested number of cores by 1. We chose this approach because load factor is a fairly intuitive proxy for response time; this makes it easier for users to specify a non-default value if needed. In addition, performance measurements showed that load factor works better than utilization for scaling up: a single load factor threshold works for a variety of workloads, whereas the best utilization for scaling up depends on the burstiness and overall volume of the workload.

CloudLab m510[36]	
CPU	Xeon D-1548 (8 x 2.0 GHz cores)
RAM	64 GB DDR4-2133 at 2400 MHz
Disk	Toshiba THNSN5256GPU7 (256 GB)
NIC	Dual-port Mellanox ConnectX-3 10 Gb
Switches	HPE Moonshot-45XGc

Table 1: Hardware configuration used for benchmarking. All nodes ran Linux 4.4.0. C-States were enabled and Meltdown mitigations were disabled. Hyperthreads were enabled (2 hyperthreads per core). Machines were not configured to perform packet steering such as RSS or XPS.

On the other hand, scaling down is based on utilization. Load factor is hard to use for scaling down because the metric of interest is not the current load factor, but rather the load factor that will occur with one fewer core; this is hard to estimate. Instead, the default core policy records the total utilization (sum of the utilizations of all cores running normal threads) each time it increases its requested number of cores. When the utilization returns to a level slightly less than this, the runtime reduces its requested number of cores by 1 (the "slightly less" factor provides hysteresis to prevent oscillations). A separate scale-down utilization is recorded for each distinct number of requested cores.

Overall, three parameters control the core estimation mechanism: the load factor for scaling up, the interval over which statistics are averaged for core estimation, and the hysteresis factor for scaling down. The default core policy currently uses a load factor threshold of 1.5, an averaging interval of 50 ms, and a hysteresis factor of 9% utilization.

7 Evaluation

We implemented Arachne in C++ on Linux; source code is available on GitHub [33]. The core arbiter contains 4500 lines of code, the runtime contains 3400 lines, and the default core policy contains 270 lines.

Our evaluation of Arachne addresses the following questions:

- How efficient are the Arachne threading primitives, and how does Arachne compare to other threading systems?
- Does Arachne's core-aware approach to threading produce significant benefits for low-latency applications?

Benchmark	Arachne		Arachne RQ		std::thread	Go	uThreads
	No HT	HT	No HT	HT			
Thread Creation	275 ns	320 ns	524 ns	520 ns	13329 ns	444 ns	6132 ns
One-Way Yield	83 ns	149 ns	137 ns	199 ns	N/A	N/A	79 ns
Null Yield	14 ns	23 ns	13 ns	24 ns	N/A	N/A	6 ns
Condition Notify	251 ns	272 ns	459 ns	471 ns	4962 ns	483 ns	4976 ns
Signal	233 ns	254 ns	N/A	N/A	N/A	N/A	N/A
Thread Exit Turnaround	328 ns	449 ns	408 ns	484 ns	N/A	N/A	N/A

Table 2: Median cost of scheduling primitives. Creation, notification, and signaling are measured end-to-end, from initiation in one thread until the target thread wakes up and begins execution on a different core. Arachne creates all threads on a different core from the parent. Go always creates Goroutines on the parent's core. uThreads uses a round-robin approach to assign threads to cores; when it chooses the parent's core, the median cost drops to 250 ns. In "One-Way Yield", control passes from the yielding thread to another runnable thread on the same core. In "Null Yield", there are no other runnable threads, so control returns immediately to the yielding thread. "Thread Exit Turnaround" measures the time from the last instruction of one thread to the first instruction of the next thread to run on a core. N/A indicates that the threading system does not expose the measured primitive. "Arachne RQ" means that Arachne was modified to use a ready queue instead of the queueless dispatch mechanism described in Section 5.3. "No HT" means that each thread ran on a separate core using one hyperthread; the other hyperthread of each core was inactive. "HT" means the other hyperthread of each core was active, running the Arachne dispatcher.

- How do Arachne's internal mechanisms, such as its queue-less approach to thread scheduling and its mechanisms for core estimation and core allocation, contribute to performance?

Table 1 describes the configuration of the machines used for benchmarking.

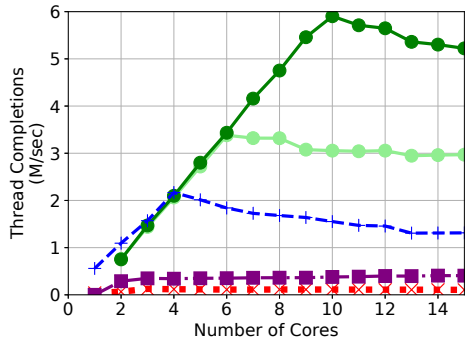
7.1 Threading Primitives

Table 2 compares the cost of basic thread operations in Arachne with C++ `std::thread`, Go, and uThreads [4]. `std::thread` is based on kernel threads; Go implements threads at user level in the language runtime; and uThreads uses kernel threads to multiplex user threads, like Arachne. uThreads is a highly rated C++ user threading library on GitHub and claims high performance. The measurements use microbenchmarks, so they represent best-case performance.

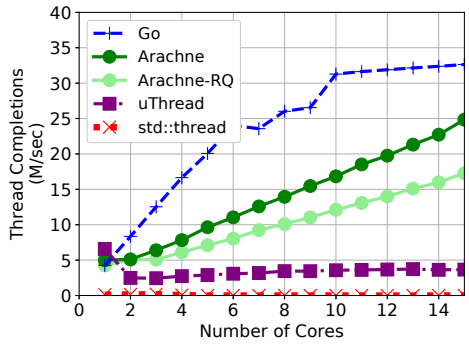
Arachne's thread operations are considerably faster than any of the other systems, except that yields are faster in uThreads. Arachne's cache-optimized design performs thread creation twice as fast as Go, even though Arachne places new threads on a different core from the parent while Go creates new threads on the parent's core.

To evaluate Arachne's queueless approach, we modified Arachne to use a wait-free multiple-producer-single-consumer queue [7] on each core to identify runnable threads instead of scanning over all contexts. We selected this implementation for its speed and simplicity from several candidates on GitHub. Table 2 shows that the queueless approach is 28–40% faster than one using a ready queue (we counted three additional cache misses for thread creation in the ready queue variant of Arachne).

We designed Arachne's thread creation mechanism not just to minimize latency, but also to provide high throughput. We ran two experiments to measure thread creation throughput. In the first experiment (Figure 2(a)), a single "dispatch" thread creates new threads as quickly as possible (this situation might occur, for example, if a single thread is polling a network interface for incoming requests). A single Arachne thread can spawn more than 5 million new threads per second, which is 2.5x the rate of Go and at least 10x the rate of `std::thread` or uThreads. This experiment demonstrates the benefits of performing load balancing at thread creation time. Go's



(a)

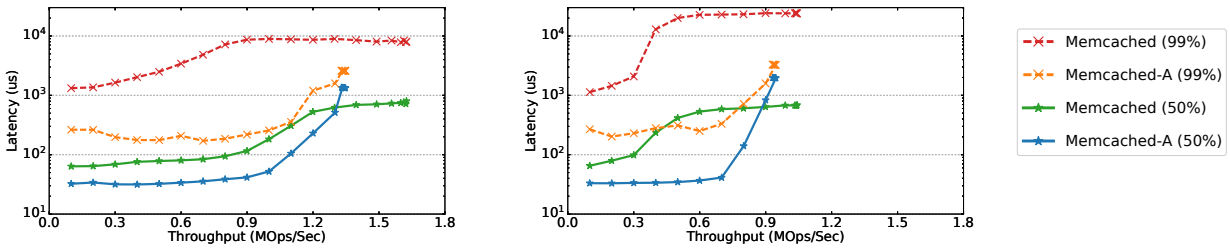


(b)

Figure 2: Thread creation throughput as a function of available cores. In (a) a single thread creates new threads as quickly as possible; each child consumes 1 µs of execution time and then exits. In (b) 3 initial threads are created for each core; each thread creates one child and then exits.

Experiment	Program	Keys	Values	Items	PUTs	Clients	Threads	Conns	Pipeline	IR Dist
Realistic	Mutilate [27]	ETC	ETC	1M	.03	20+1	16+8	1280+8	1+1	GPareto
Colocation	Memtier [24]	30B	200B	8M	0	1+1	16+8	320+8	10+1	Poisson
Skew	Memtier	30B	200B	8M	0	1	16	512	100	Poisson

Table 3: Configurations of memcached experiments. Program is the benchmark program used to generate the workload (our version of Memtier is modified from the original). Keys and Values give sizes of keys and values in the dataset (ETC recreates the Facebook ETC workload [3], which models actual usage of memcached). Items is the total number of objects in the dataset. PUTs is the fraction of all requests that were PUTs (the others were GETs). Clients is the total number of clients (20+1 means 20 clients generated an intensive workload, and 1 additional client measured latency using a lighter workload). Threads is the number of threads per client. Conns is the total number of connections per client. Pipeline is the maximum number of outstanding requests allowed per connection before shedding workload. IR Dist is the inter-request time distribution. Unless otherwise indicated, memcached was configured with 16 worker threads and memcached-A scaled automatically between 2 and 15 cores.



(a) memcached: 16 worker threads, 16 cores **(b)** memcached: 16 workers; both: 8 cores

Figure 3: Median and 99th-percentile request latency as a function of achieved throughput for both memcached and memcached-A, under the Realistic benchmark. Each measurement ran for 30 seconds after a 5-second warmup. Y-axes use a log scale.

work stealing approach creates significant additional overhead, especially when threads are short-lived, and the parent's work queue can become a bottleneck. At low core counts, Go exhibits higher throughput than Arachne because it runs threads on the creator's core in addition to other available cores, while Arachne only uses the creator's core for dispatching.

The second experiment measures thread creation throughput using a distributed approach, where each of many existing threads creates one child thread and then exits (Figure 2(b)). In this experiment both Arachne and Go scaled in their throughput as the number of available cores increased. Neither uThreads nor `std::thread` had scalable throughput; uThreads had 10x less throughput than Arachne or Go and `std::thread` had 100x less throughput. Go's approach to thread creation worked well in this experiment; each core created and executed threads locally and there was no need for work stealing since the load naturally balanced itself. As a result, Go's throughput was 1.5–2.5x that of Arachne. Arachne's performance reflects the costs of thread creation and exit turnaround from Table 2, as well as occasional conflicts between concurrent thread creations.

Figure 2 also includes measurements of the ready queue variant of Arachne. Arachne's queueless approach provided higher throughput than the ready queue variant for both experiments.

7.2 Arachne's benefits for memcached

We modified memcached [23] version 1.5.6 to use Arachne; the source is available on GitHub [19]. In the modified version ("memcached-A"), the pool of worker threads is replaced by a single dispatch thread, which waits for incoming requests on all connections. When a request arrives, the dispatch thread creates a new Arachne thread, which lives only long enough to handle all available requests on the connection. Memcached-A uses the default core policy; the dispatch thread is "exclusive" and workers are "normal" threads.

Memcached-A provides two benefits over the original memcached. First, it reduces performance interference, both between kernel threads (there is no multiplexing) and between applications (cores are dedicated to applications). Second, memcached-A provides finer-grain load-balancing (at the level of individual requests rather than connections).

We performed three experiments with memcached; their configurations are summarized in Table 3. The first experiment, Realistic, measures latency as a function of load under realistic conditions; it uses the Mutilate benchmark [17, 27] to recreate the Facebook ETC workload [3]. Figure 3(a) shows the results. The maximum throughput of memcached is 20% higher than memcached-A. This is because memcached-A uses two fewer cores (one core is reserved for unmanaged threads and one for the dispatcher); in addition, memcached-A incurs overhead to spawn a thread for each request. However, memcached-A has significantly lower latency than memcached. Thus, if an application has service-level requirements, memcached-A provides higher useable throughput. For example, if an application requires

a median latency less than 100 µs, memcached-A can support 37.5% higher throughput than memcached (1.1 Mops/sec vs. 800 Kops/sec). At the 99th percentile, memcached-A's latency ranges from 3–40x lower than memcached. We found that Linux migrates memcached threads between cores frequently: at high load, each thread migrates about 10 times per second; at low load, threads migrate about every third request. Migration adds overhead and increases the likelihood of multiplexing.

One of our goals for Arachne is to adapt automatically to application load and the number of available cores, so administrators do not need to specify configuration options or reserve cores. Figure 3(b) shows memcached's behavior when it is given fewer cores than it would like. For memcached, the 16 worker threads were multiplexed on only 8 cores; memcached-A was limited to at most 8 cores. Maximum throughput dropped for both systems, as expected. Arachne continued to operate efficiently: latency was about the same as in Figure 3(a). In contrast, memcached experienced significant increases in both median and tail latency, presumably due to additional multiplexing; with a median latency limit of 100 µs, memcached could only handle 300 Kops/sec, whereas memcached-A handled 780 Kops/sec.

The second experiment, Colocation, varied the load dynamically to evaluate Arachne's core estimator. It also measured memcached and memcached-A performance when colocated with a compute-intensive application (the x264 video encoder [25]). The results are in Figure 4. Figure 4(a) shows that memcached-A used only 2 cores at low load (dispatch and one worker) and ramped up to use all available cores as the load increased. Memcached-A maintained near-constant median and tail latency as the load increased, which indicates that the core estimator chose good points at which to change its core requests. Memcached's latency was higher than memcached-A and it varied more with load; even when running without the background application, 99th-percentile latency was 10x higher for memcached than for memcached-A. Tail latency for memcached-A was actually better at high load than low load, since there were more cores available to absorb bursts of requests.

When a background video application was colocated with memcached, memcached's latency doubled, both at the median and at the 99th percentile, even though the background application ran at lower priority. In contrast, memcached-A was almost completely unaffected by the video application. This indicates that Arachne's core-aware approach improves performance isolation between applications. Figure 4(a) shows that memcached-A ramped up its core usage more quickly when colocated with the video application. This suggests that there was some performance interference from the video application, but that the core estimator detected this and allocated cores more aggressively to compensate.

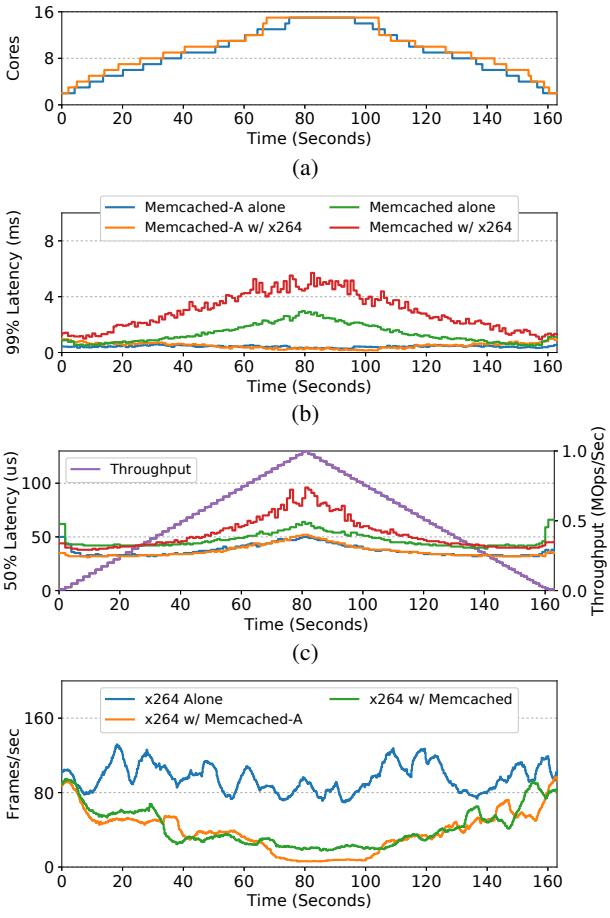


Figure 4: Memcached performance in the Colocation experiment. The request rate increased gradually from 10 Kops/sec to 1 Mops/sec and then decreased back to 10 Kops/sec. In some experiments the x264 video encoder [25] ran concurrently, using a raw video file (sintel-1280.y4m) from Xiph.org [21]. When memcached-A ran with x264, the core arbiter gave memcached-A as many cores as it requested; x264 was not managed by Arachne, so Linux scheduled it on the cores not used by memcached-A. When memcached ran with x264, the Linux scheduler determined how many cores each application received. x264 sets a "nice" value of 10 for itself by default; we did not change this behavior in these experiments. (a) shows the number of cores allocated to memcached-A; (b) shows 99th percentile tail latency for memcached and memcached-A; (c) shows median latency, plus the rate of requests; (d) shows the throughput of the video decoder (averaged over trailing 4 seconds) when running by itself or with memcached or memcached-A.

Figure 4(d) shows the throughput of the video application. At high load, its throughput when colocated with memcached-A was less than half its throughput when colocated with memcached. This is because memcached-A confined the video application to a single unmanaged core. With memcached, Linux allowed the video appli-

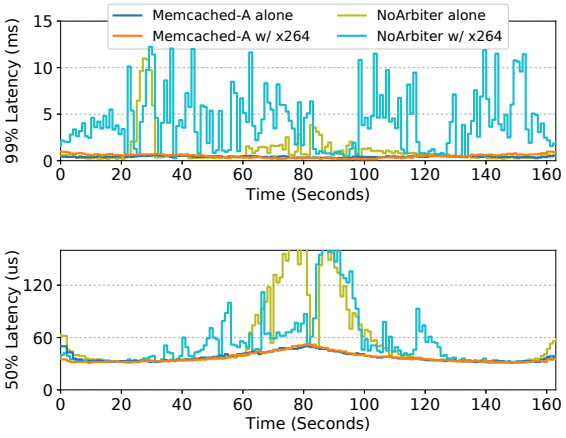


Figure 5: Median and tail latency in the Colocation experiment with the core arbiter ("Memcached-A", same as Figure 4) and without the core arbiter ("NoArbiter").

cation to consume more resources, which degraded the performance of memcached.

Figure 5 shows that dedicated cores are fundamental to Arachne's performance. For this figure, we ran the Colocation experiment using a variant of memcached-A that did not have dedicated cores: instead of using the core arbiter, Arachne scaled by blocking and unblocking kernel threads on semaphores, and the Linux kernel scheduled the unblocked threads. As shown in Figure 5, this resulted in significantly higher latency both with and without the background application. Additional measurements showed that latency spikes occurred when Linux descheduled a kernel thread but Arachne continued to assign new user threads to that kernel thread; during bursts of high load, numerous user threads could be stranded on a descheduled kernel thread for many milliseconds. Without the dedicated cores provided by the core arbiter, memcached-A performed significantly worse than unmodified memcached.

Figures 4 and 5 used the default configuration of the background video application, in which it lowered its execution priority to "nice" level 10. We also ran the experiments with the video application running at normal priority; median and tail latencies for memcached increased by about 2x, while those for memcached-A were almost completely unaffected. We omit the details, due to space limitations.

The final experiment for memcached is Skew, shown in Figure 6. This experiment evaluates memcached performance when the load is not balanced uniformly across client connections. Since memcached statically parititions client connections among worker threads, hotspots can develop, where some workers are overloaded while others are idle; this can result in poor overall throughput. In contrast, memcached-A performs load-balancing on each request, so performance is not impacted by the distribution

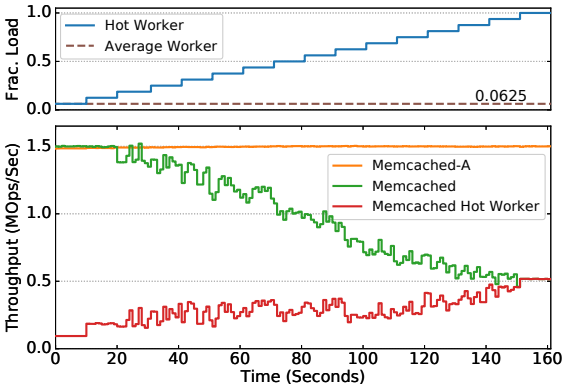


Figure 6: The impact of workload skew on memcached performance with a target load of 1.5 Mops/sec. Initially, the load was evenly distributed over 512 connections (each memcached worker handled 512/16 = 32 connections); over time, an increasing fraction of total load was directed to one specific "hot" worker thread by increasing the request rate on the hot worker's connections and decreasing the request rate on all other connections. The bottom graph shows the overall throughput, as well as the throughput of the overloaded worker thread in memcached.

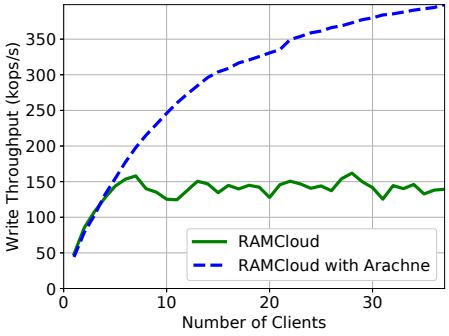


Figure 7: Throughput of a single RAMCloud server when many clients perform continuous back-to-back write RPCs of 100-byte objects. Throughput is measured as the number of completed writes per second.

of load across client connections.

7.3 Arachne's Benefits for RAMCloud

We also modified RAMCloud [30] to use Arachne. In the modified version ("RAMCloud-A"), the long-running pool of worker threads is eliminated, and the dispatch thread creates a new worker thread for each request. Threads that are busy-waiting on nested RPCs yield after each iteration of their polling loop. This allows other requests to be processed during the waiting time, so that the core isn't wasted. Figure 7 shows that RAMCloud-A has 2.5x higher write throughput than RAMCloud. On the YCSB benchmark [9] (Figure 8), RAMCloud-A provided 54% higher throughput than RAMCloud for the write-heavy YCSB-A workload. On the read-only YCSB-C workload, RAMCloud-A's throughput was 15% less than RAMCloud, due to the overhead of Arachne's thread invocation and thread exit. These experiments demonstrate

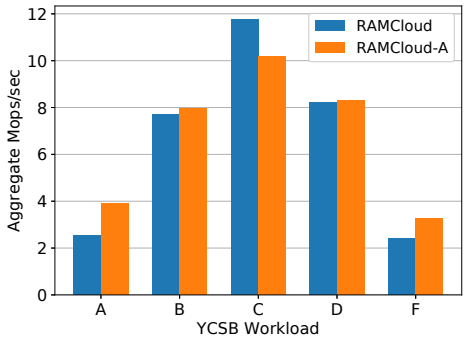


Figure 8: Comparison between RAMCloud and RAMCloud-A on a modified YCSB benchmark [9] using 100-byte objects. Both were run with 12 storage servers. Y-values represent aggregate throughput across 30 independent client machines, each running with 8 threads.

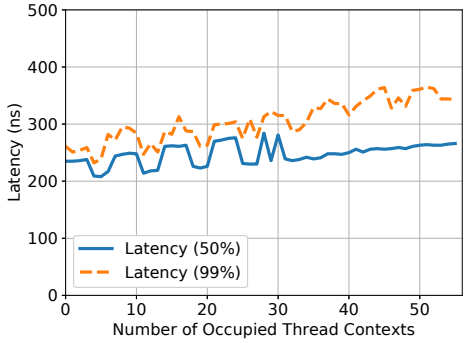


Figure 9: Cost of signaling a blocked thread as the number of threads on the target core increases. Latency is measured from immediately before signaling on one core until the target thread resumes execution on a different core.

that Arachne makes it practical to schedule other work during blockages as short as a few microseconds.

7.4 Arachne Internal Mechanisms

This section evaluates several of the internal mechanisms that are key to Arachne's performance. As mentioned in Section 5.3, Arachne forgoes the use of ready queues as part of its cache-optimized design; instead, the dispatcher scans the `wakeupTime` variables for occupied thread contexts until it finds a runnable thread. Consequently, as a core fills with threads, its dispatcher must iterate over more and more contexts. To evaluate the cost of scanning these flags, we measured the cost of signaling a particular blocked thread while varying the number of additional blocked threads on the target core; Figure 9 shows the results. Even in the worst case where all 56 thread contexts are occupied, the average cost of waking up a thread increased by less than 100 ns, which is equivalent to about one cache coherency miss. This means that an alternative implementation that avoids scanning all the active contexts must do so without introducing any new cache misses; otherwise its performance will be worse than Arachne. Arachne's worst-case performance in Figure 9 is still better than the ready queue variant of Arachne in Table 2.

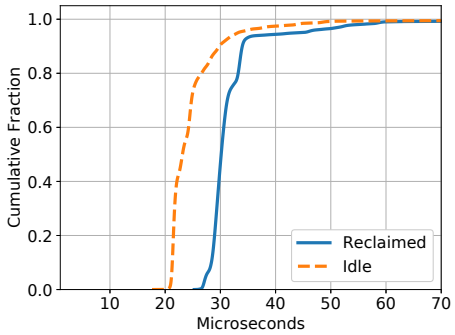


Figure 10: Cumulative distribution of latency from core request to core acquisition (a) when the core arbiter has a free core available and (b) when it must reclaim a core from a competing application.

Figures 10 and 11 show the performance of Arachne's core allocation mechanism. Figure 10 shows the distribution of allocation times, measured from when a thread calls `setRequestedCores` until a kernel thread wakes up on the newly-allocated core. In the first scenario, there is an idle core available to the core arbiter, and the cost is merely that of moving a kernel thread to the core and unblocking it. In the second scenario, a core must be reclaimed from a lower priority application so the cost includes signaling another process and waiting for it to release a core. Figure 10 shows that Arachne can reallocate cores in about 30 µs, even if the core must be reclaimed from another application. This makes it practical for Arachne to adapt to changes in load at the granularity of milliseconds.

Figure 11 shows the timing of each step of a core request that requires the preemption of another process's core. About 80% of the time is spent in socket communication.

8 Related Work

Numerous user-level threading packages have been developed over the last several decades. We have already compared Arachne with Go [14] and uThreads [4]. Boost fibers [1], Folly [13], and Seastar [37] implement user-level threads but do not multiplex user threads across multiple cores. Capriccio [39] solved the problem of blocking system calls by replacing them with asynchronous system calls, but it does not scale to multiple cores. Cilk [8] is a compiler and runtime for scheduling tasks over kernel threads, but does not handle blocking and is not core-aware. Carbon [16] proposes the use of hardware queues to dispatch hundred-instruction-granularity tasks, but it requires changes to hardware and is limited to a fork-join model of parallelism. Wikipedia [40] lists 21 C++ threading libraries as of this writing. Of these, 10 offer only kernel threads, 3 offer compiler-based automatic parallelization, 3 are commercial packages without any published performance numbers, and 5 appear to be defunct. None of the systems listed above supports load balancing at thread creation time, the ability to compute core require-

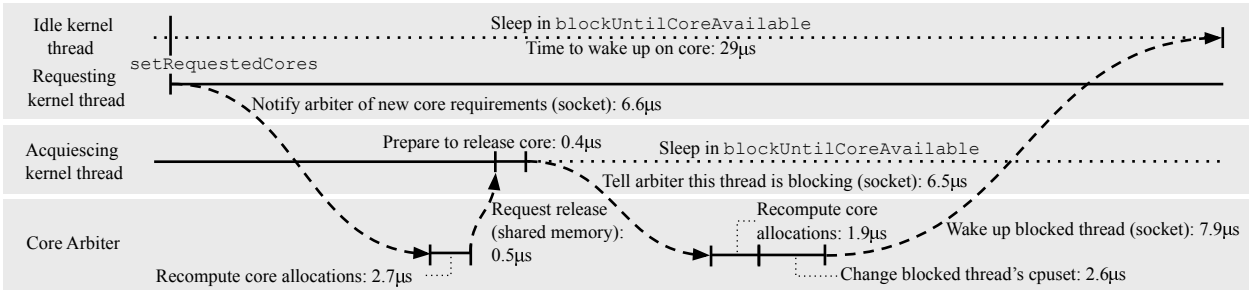


Figure 11: Timeline of a core request to the core arbiter. There are two applications. Both applications begin with a single dedicated core, and the top application also begins with a thread waiting to be placed on a core. The top application has higher priority than the bottom application, so when the top application requests an additional core the bottom application is asked to release its core.

ments and conform to core allocations, or a mechanism for implementing application-specific core policies.

Scheduler activations [2] are similar to Arachne in that they allocate processors to applications to implement user-level threads efficiently. A major focus of the scheduler activations work was allowing processor preemption during blocking kernel calls; this resulted in significant kernel modifications. Arachne focuses on other issues, such as minimizing cache misses, estimating core requirements, and enabling application-specific core policies.

Akaros [35] and Parlib [15] follow in the tradition of scheduler activations. Akaros is an operating system that allocates dedicated cores to applications and makes all blocking system calls asynchronous; Parlib is a framework for building user schedulers on dedicated cores. Akaros offers functionality analogous to the Arachne core arbiter, but it does not appear to have reached a level of maturity that can support meaningful performance measurements.

The core arbiter's controlling of process scheduling policy in userspace while leaving mechanism to the kernel resembles policy modules in Hydra [18].

The traditional approach for managing multi-threaded applications on multi-core machines has been gang scheduling [12, 29]. In gang scheduling, each application unilaterally determines its threading requirements; the operating system then attempts to schedule all of an application's threads simultaneously on different cores. Tucker and Gupta pointed out that gang scheduling results in inefficient multiplexing when the system is overloaded [38]. They argued that it is more efficient to divide the cores so that each application has exclusive use of a few cores; an application can then adjust its degree of parallelism to match the available cores. Arachne implements this approach.

Event-based applications such as Redis [34] and nginx [28] represent an alternative to user threads for achieving high throughput and low latency. Behren et al. [39] argued that event-based approaches are a form of application-specific optimization and such optimization is due to the lack of efficient thread runtimes; Arachne offers efficient threading as a more convenient alternative to events.

Several recent systems, such as IX [6] and Zygos [32], have combined thread schedulers with high-performance network stacks. These systems share Arachne's goal of combining low latency with efficient resource usage, but they take a more special-purpose approach than Arachne by coupling the threading mechanism to the network stack. Arachne is a general-purpose mechanism; it can be used with high-performance network stacks, such as in RAMCloud, but also in other situations.

9 Future Work

We believe that Arachne's core-aware approach to scheduling would be beneficial in other domains. For example, virtual machines could use a multi-level core-aware approach, where applications use Arachne to negotiate with their guest OS over cores, and the guest OSes use a similar approach to negotiate with the hypervisor. This would provide a more flexible and efficient way of managing cores than today's approaches, since the hypervisor would know how many cores each virtual machine needs.

Core-aware scheduling would also be beneficial in cluster schedulers for datacenter-scale applications. The cluster scheduler could collect information about core requirements from the core arbiters on each of the cluster machines and use this information to place applications and move services among machines. This would allow decisions to be made based on actual core needs rather than statically declared maximum requirements. Arachne's performance isolation would allow cluster schedulers to run background applications more aggressively without fear of impacting the response time of foreground applications.

There are a few aspects of Arachne that we have not fully explored. We have only preliminary experience implementing core policies, and our current core policies do not address issues related to NUMA machines, such as how to allocate cores in an application that spans multiple sockets. We hope that a variety of reusable core policies will be created, so that application developers can achieve high threading performance without having to write a cus-

tom policy for each application. In addition, our experience with the parameters for core estimation is limited. We chose the current values based on a few experiments with our benchmark applications. The current parameters provide a good trade-off between latency and utilization for our benchmarks, but we don't know whether these will be the best values for all applications.

10 Conclusion

One of the most fundamental principles in operating systems is virtualization, in which the system uses a set of physical resources to implement a larger and more diverse set of virtual entities. However, virtualization only works if there is a balance between the use of virtual objects and the available physical resources. For example, if the usage of virtual memory exceeds available physical memory, the system will collapse under page thrashing.

Arachne provides a mechanism to balance the usage of virtual threads against the availability of physical cores. Each application computes its core requirements dynamically and conveys that to a central core arbiter, which then allocates cores among competing applications. The core arbiter dedicates cores to applications and tells each application which cores it has received. The application can then use that information to manage its threads. Arachne also provides an exceptionally fast implementation of threads at user level, which makes it practical to use threads even for very short-lived tasks. Overall, Arachne's core-aware approach to thread management enables granular applications that combine both low latency and high throughput.

11 Acknowledgements

We would like to thank our shepherd, Michael Swift, and the anonymous reviewers for helping us improve this paper. Thanks to Collin Lee for giving feedback on design, and to Yilong Li for help with debugging the RAMCloud networking stack. This work was supported by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA) and by the industrial affiliates of the Stanford Platform Lab.

References

[1] Boost fibers. `http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/fiber/overview.html`.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[4] S. Barghi. uthreads: Concurrent user threads in c++. `https://github.com/samanbarghi/uThreads`.

[5] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, Mar. 2017.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Oct. 2014.

[7] D. Bittman. Mpscq - multiple producer, single consumer wait-free queue. `https://github.com/dbittman/waitfree-mpsc-queue`.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, January 2008.

[11] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[12] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[13] Folly: Facebook open-source library. `https://github.com/facebook/folly`.

[14] The Go Programming Language. `https://golang.org/`.

[15] K. A. Klues. *OS and Runtime Support for Efficiently Managing Cores in Parallel Applications*. PhD thesis, University of California, Berkeley, 2015.

[16] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, 2007.

[17] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, 2014.

[18] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 132–140. ACM, 1975.

[19] Q. Li. memcached-a. `https://github.com/PlatformLab/memcached-A`.

[20] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Apr. 2014.

[21] M. Lora. Xiph.org::test media. `https://media.xiph.org/`.

[22] A. Lottarini, A. Ramirez, J. Coburn, M. A. Kim, P. Ranganathan, D. Stodolsky, and M. Wachsler. vbench: Benchmarking video transcoding in the cloud. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–809, 2018.

[23] memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`.

[24] Memtier benchmark. `https://github.com/RedisLabs/memtier_benchmark`.

[25] L. Merritt and R. Vanam. x264: A high performance H.264/AVC encoder. `http://neuron2.net/library/avc/overview_x264_v8_5.pdf`.

[26] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[27] Mutilate: high-performance memcached load generator. `https://github.com/leverich/mutilate`.

[28] Nginx. `https://nginx.org/en/`.

[29] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[30] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[31] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.

[32] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proc. of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, 2017.

[33] H. Qin. Arachne. `https://github.com/PlatformLab/Arachne`.

[34] Redis. `http://redis.io`.

[35] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, 2011.

[36] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[37] Seastar. `http://www.seastar-project.org/`.

[38] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 159–166, 1989.

[39] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 268–281, 2003.

[40] Wikipedia. List of c++ multi-threading libraries — wikipedia, the free encyclopedia, 2017.

Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models

Suli Yang*, Jing Liu†, Andrea C. Arpaci-Dusseau†, Remzi H. Arpaci-Dusseau†

Ant Financial Services Group* University of Wisconsin-Madison†

Abstract

In this paper, we present an approach to systematically examine the *schedulability* of distributed storage systems, identify their scheduling problems, and enable effective scheduling in these systems. We use *Thread Architecture Models (TAMs)* to describe the behavior and interactions of different threads in a system, and show both how to construct TAMs for existing systems and utilize TAMs to identify critical scheduling problems. We identify five common problems that prevent a system from providing schedulability and show that these problems arise in existing systems such as HBase, Cassandra, MongoDB, and Riak, making it difficult or impossible to realize various scheduling disciplines. We demonstrate how to address these schedulability problems by developing Tamed-HBase and Muzzled-HBase, sets of modifications to HBase that can realize the desired scheduling disciplines, including fairness and priority scheduling, even when presented with challenging workloads.

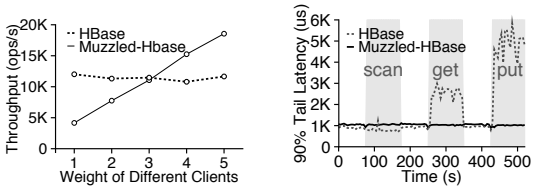
1 Introduction

The modern data center is built atop massive, scalable storage systems [12, 25, 42, 51]. For example, a typical Google cluster consists of tens of thousands of machines, with PBs of storage spread across hard disk drives (or SSDs) [51]. These expansive storage resources are managed by Colossus, a second-generation scalable file system that replaced the original GFS [25]; many critical Google applications (e.g., Gmail and Youtube), as well as generic cloud-based services, co-utilize Colossus and thus contend for cluster-wide storage resources such as disk space and I/O bandwidth.

As a result, a critical aspect of these storage systems is how they *share* resources. If, for example, requests from one application can readily drown out requests from another, building scalable and predictable applications and services becomes challenging (if not impossible).

To address these concerns, scalable storage systems must provide correct and efficient *request scheduling* as a fundamental primitive. By controlling which client or application is serviced, critical features including fair sharing [28, 38, 58, 66], throughput guarantees [54, 68], low tail latency [19, 29, 47, 63, 72] and performance isolation [9, 55, 62] can be successfully realized.

Unfortunately, modern storage systems are complex, concurrent programs. Many systems are realized via an intricate series of stages, queues, and thread pools, based loosely on SEDA design principles [64]. For example, HBase [24] consists of ~500K lines of code, and involves ~1000 interacting threads within each server when running. Understanding how to introduce scheduling control into systems is challenging even for those who develop them; a single request may flow through numerous stages across multiple machines while being serviced.



(a) **Weighted Fair Share**: *Original HBase does not respect different weights of clients, yet in Muzzled-HBase all the clients get throughput proportional to their weights.*

(b) **Latency Guarantee**: *The tail latency of the foreground client is severely impacted by background workloads (cached scan, random get, and random put, indicated by grey bars) with original HBase, but remains stable with Muzzled-HBase.*

Figure 1: **TAM Enable SLOs (HBase).** *Muzzled-HBase supports multiple scheduling policies under YCSB benchmark. Experiment setup described in §4.2.*

All of the open-source storage systems we examined have significant scheduling deficiencies, thus rendering them unable to achieve desired scheduling goals. As shown in Figure 1, the original HBase fails to provide weighted fairness or isolation against background workloads, yet our implemenation of Muzzled-HBase successfully achieved these goals. Such scheduling deficiencies have also caused significant problems in production, including extremely low write throughput or even data loss for HBase [5], unbounded read latency for MongoDB [6, 7], and imbalance between workloads in Cassandra [4]. All above problems have been assigned major or higher priority by the developers, but remain unsolved due to their complexities and the amount of changes required to the systems.

To remedy this problem, and to make the creation of flexible and effective scheduling policies within complex storage systems easy, this paper presents a novel approach to such *schedulability analysis*, which allows systematic reasoning on how well a system could support scheduling based on its thread architecture. Specifically, we define a *Thread Architecture Model* (TAM), which captures the behavior and interactions of different threads within a system. By revealing the resource

* Work done while at University of Wisconsin-Madison.

consumption patterns and dependencies between components, a TAM effectively links the performance of a storage system to its architecture (while abstracting away implementation details). Using a TAM, various scheduling problems can be discerned, pointing toward solutions that introduce necessary scheduling controls. The system can then be transformed to provide schedulability by fixing these problems, allowing realization of various scheduling policies atop it. TAMs are also readily visualized using *Thread Architecture Diagrams* (TADs), and can be (nearly) automatically obtained by tracing a system of interest under different workloads.

We use TAMs to analyze the schedulability of four important and widely-used scalable storage systems: HBase/HDFS [24, 56], Cassandra [36], MongoDB [15], and Riak [33], and highlight weaknesses in the scheduling architecture of each. Our analysis centers around five essential problems we have discovered, each of which leads to inadequate scheduling controls: a lack of local scheduling control points, unknown resource usage, hidden competition between threads, uncontrolled thread blocking, and ordering constraints upon requests. Fortunately, these problems can be precisely specified using TAMs, enabling straightforward and automatic problem identification. These problems can also be visually identified using TADs, allowing system architects to readily understand where problems arise.

By fixing the problems identified using TAM, HBase, the most complex system we studied, can be transformed to provide schedulability. We show via simulation that Tamed-HBase (TAM-EnableD HBase) utilizes a problem-free thread architecture to enable fair sharing under intense resource competition and provide strong tail latency guarantees with background interference; it also achieves proper isolation despite variances in request amount, size, and other workload factors. We implement Muzzled-HBase (an approximation of Tamed-HBase) to show that TAM-guided schedulability analysis corresponds to the real world.

The rest of this paper is structured as follows. We first introduce the thread architecture model (TAM) (§2), and then discuss how to use TAM to perform schedulability analysis, centered around the five scheduling problems (§3). We use HBase/HDFS as a case study to demonstrate how to use TAM to analyze the schedulability of a realistic system, and make said system schedulable (§4). We then present the schedulability analysis results of other systems (§5). Next, we discuss the limitations of TAM and how it can be extended (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Thread Architecture Model

Implementing new scheduling policies in existing systems is non-trivial; most modern scalable storage systems have complex structures with specific features that complicate the realization of scheduling policies. We introduce *thread architecture models* (TAMs) to describe these structures. The advantage of TAM is that *one can perform schedulability analysis with only information specified in this model, abstracting away all the implementation details.* We first give a general and intuitive description of TAM (§2.1) and describe its visualization using TAD (§2.2). We then discuss how to automatically obtain TAM for existing systems (§2.3). Finally, we give a formal definition of the TAM model (§2.4).

2.1 TAM: General Description

We model scheduling in a storage system as containing requests that flow through the data path consuming various resources while a control plane collects information and determines a scheduling plan to realize the system's overall goal (e.g., fairness). This plan is enforced by local schedulers at different points along the data path.

In modern SEDA-based distributed storage systems, the data path consists of many distinct *stages* residing in different *nodes*. A stage contains threads performing similar tasks (e.g., handling RPC requests or performing I/O). A *thread* refers to any sequential execution (e.g., a kernel thread, a user-level thread, or a virtual process in a virtual machine). Within a stage, threads can be organized as a pool with a fixed (or maximum) number of active threads (*bounded stage*) or can be allocated dynamically as requests increase (*on-demand stage*). In certain stages, some requests may need to be served in a specific order for correctness; this is an *ordering constraint*.

Each bounded stage has an associated queue from which threads pick tasks; each queue is a potential *scheduling point* where schedulers can reorder requests. The queue can be either implicit (e.g., the default FIFO queue of a Java thread pool) or explicit (with an API to allow choice of policy, or hard-coded decisions).

A stage may pass requests to its *downstream* stages for processing. After a thread issues a request to downstream stages, the thread may immediately proceed to another request, or *block* until notified that the request completed at other stages.

Resources are consumed within a stage as requests are processed; we consider CPU, I/O, network and lock[1] resources, but other resources can be readily added to our model. Instead of specifying the exact amount of resources used in each stage (which can change based on specific characteristics of workloads), we only consider whether a resource is *extensively* used in a stage. This simplification allows us to abstract away the details of slightly different workloads but still captures important problems related to resource usage (shown in §3). Exten-

[1] We treat each lock instance as a separate resource, but are usually only interested in one or two highly contended locks in the system, e.g., the namespace lock in HDFS.

C [] N L name	stage [Boxes above: its resource vector. Stop: ordering constraints]
	Scheduling point [Plug: allows pluggable schedulers. No scheduling point: on-demand stage]
A → B	Downstream relationship [Stage A issues requests to stage B]
A ⇠ B	Blocking relationship [Stage A blocks on the stage B]
	Node boundary
C [] N L	CPU, I/O, network, lock resource [Left to right. Square bracket: unknown usage]

Table 1: **Notation for Thread Architecture Diagrams.**

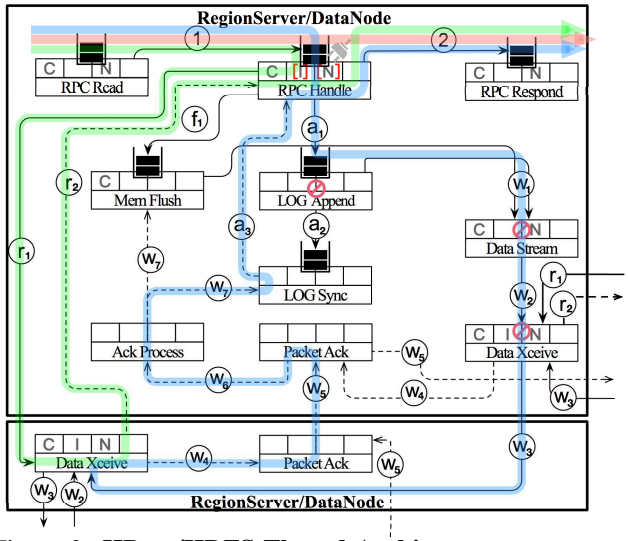


Figure 2: **HBase/HDFS Thread Architecture.** *Based on HBase* 2.0.0 *and Hadoop* 2.7.1. *Some stages are omitted to simplify discussion. Red: main RPC processing flow; green: processing flow that requires HDFS read; blue: processing flow that requires data modifications.*

sive resource usage is interpreted as "any serious usage of resources that is worth considering during scheduling"; we discuss how we choose the threshold in §2.3. If a stage may or may not extensively use a resource during processing based on different workloads, it has *unknown resource usage* for this resource.

All the stages and their collective behaviors, relationships, and resource consumption patterns form the *thread architecture* of a system.

2.2 Visualization with TAD

One advantage of TAM is that it allows direct visualizations using *Thread Architecture Diagrams* (TADs). Table 1 summarizes the building blocks in TADs; Figure 2 through 5 show the TADs of HBase/HDFS [24, 56], MongoDB [15], Cassandra [36] and Riak [33] (labels on the arrows and important workload flows are manually added to aid understanding, and are not parts of TAD). TAM and TAD can be thought of as duals: TAD is the graphical representation of TAM, while TAM is the symbolic representation of TAD; one can easily transform a TAD to its underlying TAM, and vice versa.

We now use the (simplified) HBase/HDFS TAD in Figure 2 to illustrate how to read a TAD and identify specific features of system scheduling from it.

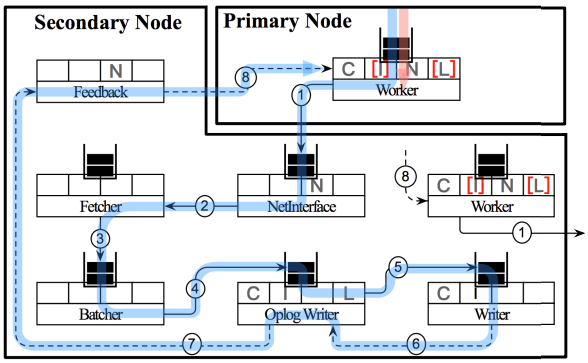


Figure 3: **MongoDB Thread Architecture.** *(v3.2.10) Red: processing flow for requests that do not block on replication; blue: processing flow for requests that do.*

When HBase clients send queries to the RegionServer, the RPC Read stage reads from the network and passes the request to the RPC Handle stage (1). Based on the request type (Put or Get) and whether data is cached, RPC Handle may have different behavior. One may insert custom schedulers into RPC Handle (plug symbol).

If the RPC needs to read data, RPC Handle checks if the data is local. If not, RPC Handle sends a read request to the Data Xceive stage in a Datanode and blocks ($r_1 - r_2$, where blocking is indicated by dashed r_2). If it is local, RPC Handle directly performs short-circuited reads, consuming I/O. I/O resource usage in RPC Handle is initially unknown and thus marked with a bracket.

For operations that modify data, RPC Handle appends WAL entries to a log (a_1) and blocks until the entry is persisted. LOG Append fetches WAL entries from the queue in the same order they are appended (stop symbol), and writes them to HDFS by passing data to Data Stream (w_1), which sends the data to Data Xceive ($w_2 - w_3$). All WAL entries append to the same HDFS file, so Data Stream and Data Xceive must process them in sequence. LOG Append also sends information about WAL entries to LOG Sync (a_2), which blocks (w_7) until the write path notifies it of completion (further details omitted); it then tells RPC Handle to proceed (dashed a_3). RPC Handle may also flush changes to the MemStore cache (f_1); when the cache is full, the content is written to HDFS with the same steps as with LOG Append writes ($w_1 - w_7$), though without the ordering constraint.

Finally, after RPC Handle finishes an RPC, it passes the result to RPC Respond and continues another RPC (2). In most cases, RPC Respond responds to the client, but if the connection is idle, RPC Handle bypasses RPC Respond and responds directly.

HBase has more than ten complex stages exhibiting different local behaviors (e.g., bounded vs. on-demand), resource usage patterns (e.g., unknown I/O demand), and interconnections (e.g., blocking and competing for the

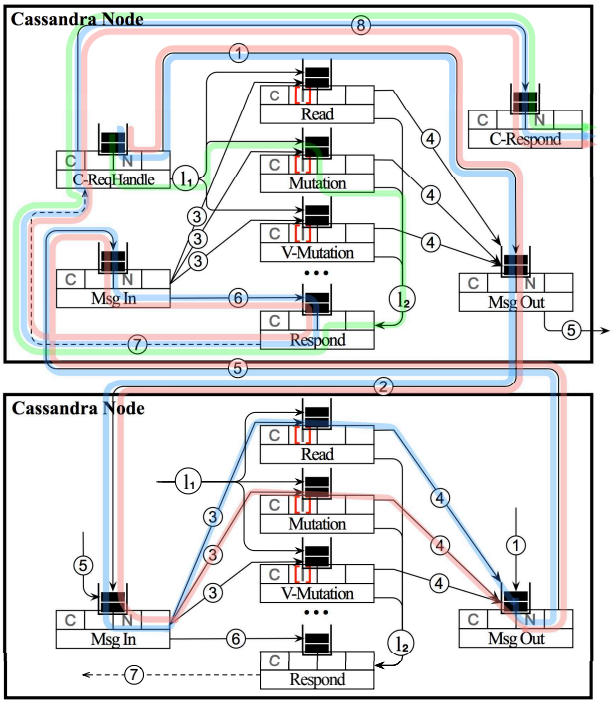


Figure 4: **Cassandra Thread Architecture.** *(v3.0.10) The ellipsis represent other database processing stages. Red: remote mutation processing flow; blue: remote read processing flow; green: local read processing flow.*

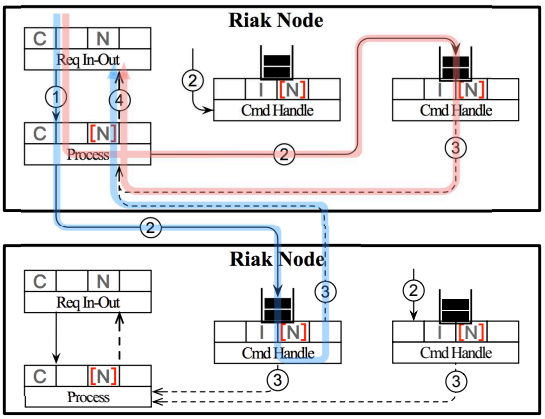


Figure 5: **Riak Thread Architecture.** *(v2.1.4) Red: local request processing flow; blue: remote request processing flow.*

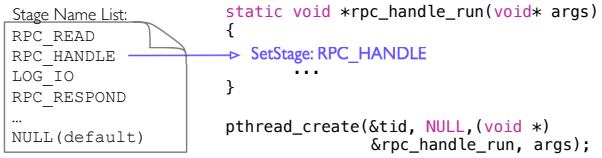


Figure 6: **Sample Annotation.**

same resources across stages). All of them are compactly encoded in its TAM/TAD, enabling us to identify problematic scheduling, as we discuss later (§3).

2.3 Automatic Obtainment

TAM is defined with automatic procurement in mind: all information specified in TAM can be (relatively) easily obtained, allowing automation of the schedulability analysis. We now present TADalyzer, a tool we developed to auto-discover TAM/TAD for real systems using instrumentation and tracing techniques. The workflow to generate the TAM of a given system with TADalyzer consists of four steps:

1. *Stage Naming*: the user lists (and names) important stages in the system.
2. *Stage Annotation*: the user identifies thread creation code in the code base and annotates if the new thread belongs to one of the stages previously named. Figure 6 shows a sample annotation. Threads not explicitly annotated default to a special `NULL` stage.
3. *Monitoring*: the user deploys the system and feeds various workloads (e.g., hot-/cold-cached, local/remote) to it. TADalyzer automatically collects necessary information for later TAM generation. If the user missed some important stages in step 1, TADalyzer would notice that some threads in the `NULL` stage are overly active, and alert the user with the stack trace of these threads. Based on the alert, the user identifies the missing stages and repeats step 1-3.
4. *Generating*: After enough information is collected, the user asks TADalyzer to generate the TAM; some information TADalyzer cannot obtain (see Figure 7), and the user needs to provide manually. TADalyzer also automatically plots TAD from the TAM (though the TADs shown in the paper are drawn manually).

This workflow requires the user to know the important (but not all) stages in the system. From our experience, someone unfamiliar with the code base usually misses naming some stages initially. However, TADalyzer provides enough information to point the user to the code of the missing stages to aid further annotation, and one can typically get a satisfactory TAM within a few (< 5) iterations of the workflow. In HBase and MongoDB such annotation took ~50 lines of code, and ~20 in Cassandra.

We now briefly describe how TADalyzer generates the TAM. Based on user annotation, TADalyzer monitors thread creation and termination, and builds a mapping between threads and stages. Using this mapping, it automatically discovers the following information:

Stage Type: TADalyzer tracks active threads at each stage to classify bounded or on-demand stages.

Resource Consumption: Using Linux kernel tracing tools [1, 40], TADalyzer attaches hooks to relevant kernel functions (e.g., `vfs_read`, `socket_read`) to monitor the I/O and network consumed at each stage. CPU consumption is tracked through */proc/stat*; the lock resource by automatically instrumenting relevant lock operations.

Intra-Node Data Flow: TADalyzer automatically instruments standard classes that are commonly used to

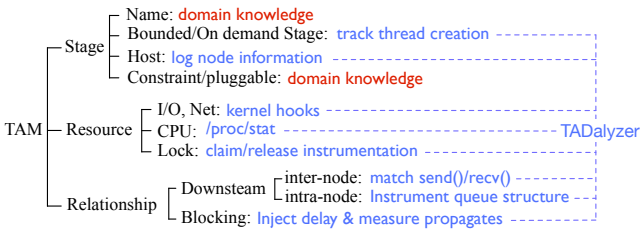


Figure 7: **TADalyzer Summary.** *Black: component in TAM, Blue: how TADalyzer obtain corresponding information in real system, Red: information TADalyzer could not provide.*

pass requests, such as `util.AbstractQueue` in Java and `std<queue>` in C++, to build data flow between stages within the same node.

Inter-Node Data Flow: TADalyzer tracks how much data each thread sends and receives on different ports. By matching the IP and port information, TADalyzer builds the data flow between stages on different nodes.

Blocking Relationship: TADalyzer injects delays in a stage and determines whether other stages block by observing if the delay propagates to these stages.

The current version of TADalyzer cannot automatically derive if a stage provides a pluggable scheduling point or has an ordering constraint, and requests this information from the user. Figure 7 summarizes how TADalyzer obtains TAM information; based on the information, TADalyzer generates the TAM/TAD.

When generating TAM, TADalyzer needs to determine the threshold for extensive resource usage of a stage. In a typical system there exist many "light" stages that are occasionally activated to perform bookkeeping and consume few resources (e.g., a timer thread); even for stages that are actively involved in request processing, they may perform tasks with a particular resource pattern and only very lightly use other resources. When accumulating the resource consumption in a long run, we observe that these stages use at most 1% of the concerned resource, while the stages that are actively consuming resources when processing requests typically use more than 10% (or much higher) of the resource. For example, in MongoDB the Worker stage consumes up to 95% of the total CPU time, while the Fetcher stage consumes at most 0.2%; similarly, in HBase the RPC Respond stage is responsible for 20% to 80% of the total bytes transferred through network, but its I/O consumption never exceeds 1%. TADalyzer thus chooses the extensiveness threshold to be within 1% and 10% to prevent these "light" stages from unnecessarily complicating the TAM (the exact threshold is set to 5%).

TADalyzer has certain limitations (detailed discussion omitted for brevity); in particular, the TAMs generated by TADalyzer are correct, but maybe incomplete (missing stages or flows).[2] However, we would like to emphasize that *TAM defines a clear set of obtainable information* (see §2.4), which *enables* tools that automatically extract this information to construct TAM and perform schedulability analysis. TADalyzer is just one such tool we built to demonstrate the feasibility of automatic schedulability analysis; we encourage other tools to be developed that deploy different techniques (e.g., those in [8, 11, 14, 69]) to discover the information listed in Figure 7 and optimize the process of obtaining TAM.

2.4 TAM: Formal Definition

We now give a more formal definition of the thread architecture model, which precisely specifies the information encoded in a TAM. Such formalism is critical for both automatically constructing TAMs (§2.3) and for systematically identifying the scheduling problems (§3).

Definition 1. A *thread architecture* is defined by the 3-tuple (S, D, B), where

- S is a finite set; each element $s \in S$ is a *stage*, which is defined in Definition 2.
- D is a function that maps S to $\mathbb{P}(S)$ (the power set of S); D represents the *downstream* relationship. For example, $D(s_1) = \{s_2, s_3\}$ means s_1 issues requests to s_2 and s_3.
- B is a function that maps S to $\mathbb{P}(S)$; B represents the *blocking* relationship. For example, $B(s_1) = \{s_2\}$ means stage s_1 blocks on stage s_2.

Definition 2. A *stage* is defined by the 5-tuple (n, h, r, o, q), where

- n is a string representing the name of the stage.
- h is a positive integer indicating host ID. Stages with the same h value are on the same node.
- r is a 4-vector representing the resource usage pattern of this stage. Each component in r can take one of the three values: `true`, `false`, or `unknown`, indicating whether the corresponding resource (CPU, I/O, network, lock) is used *extensively* in this stage.
- o is a boolean value representing whether the stage has an ordering constraint or not.
- q represents the local scheduling type, and can take one of the three values: `on_demand`, `pluggable` or `general`, indicating whether the stage is on-demand, allows pluggable schedulers, or has hard-coded or implicit scheduling logic.

3 Scheduling Problems

TAM allows us to identify scheduling problems without being concerned about low-level implementation details; it also points towards solutions that introduce necessary scheduling controls. We now discuss how to perform schedulability analysis using TAM/TAD. Our analysis

[2] All TAM/TADs shown in the paper (except MongoDB) have been validated by each system's developers [21, 23, 27].

centers around five common problems we discovered in modern distributed storage systems: *no scheduling*, *unknown resource usage*, *hidden contention*, *blocking*, and *ordering constraint*. To illustrate the process clearly, we begin by focusing on systems with only a single problem; in Section 4 we consider the HBase TAM in which multiple problematic stages are interconnected.

For each problem, we first give a general description, then precisely specify the problem in TAM and TAD. We use simulation to demonstrate how problematic thread architecture hinders scheduling policy realization; different scheduling policies including fairness, latency guarantees, and priority scheduling are investigated.

The simulation framework (built on simpy [39]) provides building blocks such as requests, threads, stages, resources, and schedulers. Using TAMs as blueprints, stages can be assembled to form various thread architectures that refect existing or hypothetical system designs. With a given thread architecture, one can specify workload characteristics (e.g., request types and arrival distribution), resource configurations (e.g., CPU frequency and network bandwidth), and scheduling policies; the framework then simulates how requests flow through the stages and consume the resources, and reports detailed performance statistics.

Unless noted, all simulations in this section use a common configuration: two competing clients (C1 and C2) continuously issue requests; C1 has 40 threads, C2 varies; each node has a 1 GHz CPU, 100 MB/s disk, and a 1 Gbps network connection.

3.1 No Scheduling

Each resource-intensive stage in a thread architecture should provide local scheduling. With local scheduling for a stage, requests are explicitly queued and resource-intensive activities can be ordered according to system's overall scheduling goal. In contrast, an on-demand stage with no request queue and extensive resource usage suffers the *no scheduling* problem (e.g., the Data Stream and Data Xceive stages in HBase, and the Req In-Out and Process stages in Riak).

TAM: A TAM (S, D, B) suffers no scheduling if $\exists s \in S$, s.t. $s.r \neq$ `[false, false, false, false]` $\wedge$ $s.q =$ `on_demand`. [3]

TAD: A TAD suffers no scheduling if it contains stages with non-empty resource boxes but no queues.

Figure 8(a) shows a simple TAD with two stages, the second of which has no scheduling (an on-demand stage with intensive I/O). The scheduler for Req Handle (Q1) attempts to provide a latency guarantee to C1 using earliest-deadline-first (EDF) but is unsuccessful: as

[3] A more stringent definition may require each resource intensive stage to provide `pluggable` scheduling point to allow flexible scheduling policy realization; we opt for a looser definition here.

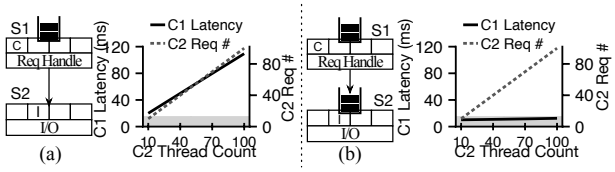


Figure 8: **The No Scheduling Problem.** *Each client request requires 100 us CPU and 100 KB I/O, making I/O the bottleneck. The deadline is set to 15 ms for C1, 500 ms for C2; the gray area indicates latency within the C1 deadline. The left y-axis shows the average latency of C1; the right y-axis shows the number of C2 requests competing with C1 at the I/O stage.*

C2 issues requests with more threads, the latency of C1 exceeds the deadline by as much as 5x. The problem occurs because Q1 scheduling is irrelevant when Req Handle is not the bottleneck: the average queue length of Q1 is zero. Meanwhile, as shown in Figure 8(a), there are many requests contending for I/O in the I/O stage, which is not managed by a scheduler.

Figure 8(b) shows another architecture which has the same functionality but does not suffer the no scheduling problem as it possesses a scheduling point at the I/O stage. Local scheduling points enable the system to regulate I/O resource usage at the point where the resource is contended, thus simply and naturally ensuring latency guarantees and isolation of the two clients.

3.2 Unknown Resource Usage

Each stage within a system should know its resource usage patterns. However, in some stages, requests may follow different execution paths with different resource usage, and these paths are not known until after the stage begins. For example, a thread could first check if a request is in cache, and if not, perform I/O; the requests in this stage have two execution paths with distinct resource patterns and the scheduler does not know this ahead of time. In such cases, the stage suffers *unknown resource usage* (e.g., the RPC Handle stage in HBase due to the short-circuited reads it might perform). Unknown resource usage forces schedulers to make decisions before information is available.

TAM: A TAM (S, D, B) suffers unknown resource usage if $\exists s \in S, \exists i \in \{1, 2, 3, 4\}$, s.t. $s.r[i] =$ `unknown`.

TAD: A TAD suffers unknown resource usage if it contains resource symbols surrounded by square brackets.

Figure 9 (a) shows a single stage with unknown I/O usage (the bracket around the I/O resource), where Q1 performs dominate resource fairness (DRF) [26] with equal weighting. When C2 issues a mix of cold and cached requests, Q1 schedules C2-cold and C2-cached in the same way. Even though there are idle CPU resources, Q1 cannot schedule additional C2-cached requests to utilize the CPU because it does not know whether the request would later cause I/O, which is currently contended. Unknown resource usage thus causes low CPU utilization and low throughput of C2-cached.

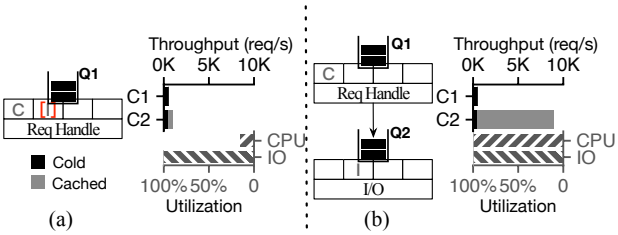


Figure 9: **The Unknown Resource Usage Problem.** *Both C1 and C2 issue requests that require 100 us CPU time and 100 KB I/O. C2 also issues cached requests that requires only 100 us CPU but no I/O. In (a) the Req Handle threads first look up the cache when serving a request, and perform I/O if it is a cache miss. In (b) a separate I/O stage performs I/O.*

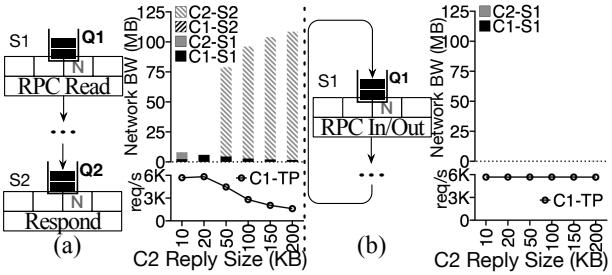


Figure 10: **The Hidden Contention Problem.** *C1 sends 1 KB requests and receives 10 KB replies; C2 also sends 1 KB requests but its reply size varies (shown in the x-axis). The line graph below shows the throughput of C1. The bar graph above shows the bandwidth each stage is forced to allocate to C1 or C2 to maintain work conservation: when scheduling, there are only C1/C2 requests in the queue. C1-S1 means the bandwidth S1 (Req Read) is forced to allocate to C1, and so on.*

Figure 9(b) shows another system with the same functionality but one stage split into two. The Req Handle stage performs CPU-intensive cache lookups while a new stage performs I/O for requests that miss the cache. Each stage has its own scheduler. Q1 freely admits requests when there are enough CPU resources, leading to high CPU utilization and C2-Cached throughput. Meanwhile, not only does Q2 know a request needs I/O, it also knows the size and location of the I/O, enabling Q2 to make better scheduling decisions. System(b) is thus free from the unknown resource usage problem.

3.3 Hidden Contention

When multiple stages with independent schedulers compete for the same resource, they suffer from *hidden contention* which impacts overall resource allocation (e.g., the Worker and Oplog Writer stages in MongoDB for database locks, and the Read, Mutation, View-Mutation stages in Cassandra for CPU and I/O). The hidden contention in MongoDB is reported to cause unbounded read latencies in production [6]. Hidden contention is ubiquitous, because some contention is difficult to avoid (e.g., most stages use CPU).

TAM: A TAM (S, D, B) suffers hidden contention if $\exists s_1 \in S$, $\exists s_2 \in S$, $\exists i \in \{1,2,3,4\}$ s.t. $s_1 \neq s_2 \wedge s_1.h = s_2.h \wedge s_1.q \neq \texttt{on_demand} \wedge s_2.q \neq \texttt{on_demand} \wedge s_1.r[i] \neq \texttt{false} \wedge s_2.r[i] \neq \texttt{false}$.

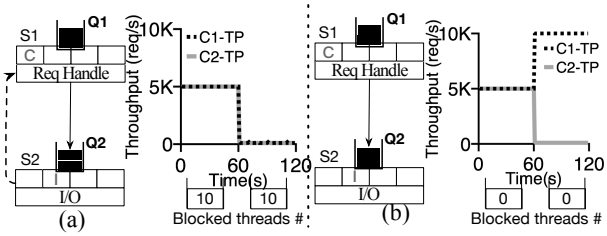


Figure 11: **The Blocking Problem.** *Initially both C1 and C2 issue requests that require 100 us CPU time and can be completed within the Req Handle Stage. At time 60, C2 switches to an I/O intensive workload where each request additionally requires 100 KB I/O at the I/O stage. C1 continues to issue CPU-only requests. The table below shows the average number of blocking threads in Req Handle (10 threads in total).*

TAD: A TAD suffers hidden contention if it contains stages within a node boundary that have separate queues but the same resource in the resource usage boxes.

Figure 10(a) shows a two-stage system with the network as the source of hidden contention; one stage reads requests and the other sends replies. Both Q1 and Q2 preform fair queuing [26] with equal weighting. However, enforcing fairness at each stage does not guarantee fair sharing at the node level. When C2 increases its reply size (i.e., its network usage), it unfairly consumes up to 95% of the network and reduces throughput of C1. With larger C2 reply size, S2 is frequently forced to schedule C2 because there are no requests from C1 in its queue. As there is no regulation on contention between stages, S2 effectively monopolizes the network when it sends larger replies (on behalf of C2) and prevents S1 from using the network; this causes fewer requests to be completed at S1 and flow to S2, further limiting the chocies available to S2. Hidden network contention between the two stages thus causes unfair scheduling.

Figure 10(b) shows a system where one stage handles both reading and replying RPCs. Q1 has full control of the network and can isolate C1 and C2 perfectly.

3.4 Blocking

For optimal performance, even when some requests are waiting to be serviced, each stage should allow other requests to make progress if possible; a problem occurs if there are no unblocked threads to serve these requests. A system has a *blocking* problem if a bounded stage may block on a downstream stage (e.g., the RPC Handle stage in HBase, and the Worker stage in MongoDB), as scenarios may occur where all threads in that stage block at one path and other requests that could have been completed cannot be scheduled. The blocking problem of HBase is reported to cause extremely low throughput or even data loss in production [5]. Blocking forces upstream schedulers to account for downstream progress.

TAM: A TAM (S, D, B) suffers blocking if $\exists s \in S$, s.t. $s.q \neq \texttt{on_demand} \wedge B(s) \neq \emptyset$.

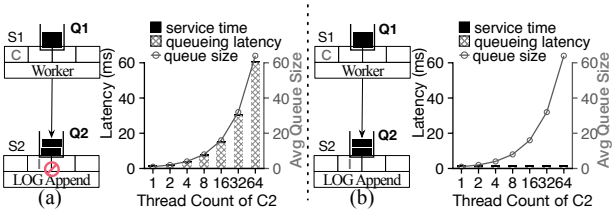


Figure 12: **The Ordering Constraint Problem.** *High priority C1 issues requests in burst; low priority C2 steadily issues requests with more threads. Each request requires 100 us CPU time at the Worker stage, and 100 KB I/O at the LOG Append stage. The left y-axis shows the average latency of C1; the right y axis shows the average queue size of LOG Append.*

TAD: A TAD suffers blocking if it contains stage boxes with queues and dashed arrows pointing to them.

Figure 11(a) shows a system with blocking at Req Handle. Requests in Req Handle have two paths: they may complete in this stage or block on the I/O stage; the schedulers perform DRF [26] with equal weighting. Initially both C1 and C2 receive high throughput as they issue cached requests without blocking; however, when C2 switches to an I/O-intensive workload, the throughput of C1 (which is still CPU-only) suffers. The table below shows that all threads in Req Handle are blocked on I/O, leaving no threads to process C1 requests.

In contrast, Figure 11(b) shows a system in which the Req Handle stage is asynchronous. No threads block; all perform useful work, leading to high throughput for C1.

3.5 Ordering Constraint

Many storage systems use Write-Ahead Logging (WAL), which requires the writes to the log to occur in sequence. When a system requires some requests at a resource-intensive stage to be served in a specific order to ensure correctness, it has the *ordering constraint* problem (e.g., the Data Stream and Data Xceive stage in HBase). Ordering constraint leaves the scheduling framework with fewer or no choices, because the local scheduler cannot reorder resource-intensive activities as desired.

TAM: A TAM (S, D, B) suffers ordering constraint if $\exists s \in S$, $\exists i \in \{1, 2, 3, 4\}$ s.t. $s.o$ = `true` $\wedge$ $s.r[i] \neq$ `false`.

TAD: A TAD suffers ordering constraint if it contains stages with stop symbols and non-empty resource boxes.

Figure 12(a) shows a two-stage system with ordering constraint on the second stage. The schedulers enforce priorities, where high priority requests are served first as long as this does not break correctness. In this system, C1 (high priority) suffers much longer latency when C2 (low priority) issues requests aggressively. The majority of this latency occurs from queuing delay in the second stage since low priority requests must be serviced first if they enter the stage earlier.

Figure 12(b) shows a system that eliminates the problem by separating requests from different clients into different streams that share no common states (e.g., each stream has its own WAL); even though requests within a stream are still serviced in order, the scheduler can choose which stream to serve and provide differentiated services on a per-stream basis. The figure shows that C1 maintains low latency despite the larger queue size at the LOG Append stage when C2 issues more requests: free from the ordering constraint, Q2 can pick the high priority requests from C1 first.

3.6 Discussion

We have identified five categories of scheduling problems. For each category, we have given an example that highlights the problem. In some cases the example highlights a fairness problem; in others it highlights a latency or utilization problem. However, one should note that each of these problems can manifest in many different ways, causing violations in any scheduling discipline. For example, in §3.1 we show how no scheduling causes excessive latencies; since there are no scheduling points to prioritize requests, it could as easily cause unfairness or priority inversions. How (and whether) the scheduling problems manifest depends on the resources available, the workload, and the scheduling policy; when TAM/TAD suggests a scheduling problem, it means that there exist certain workloads/resource configurations under which the problem manifests.

Each of the five scheduling problem by itself is not very surprising. However, by compactly representing the thread architecture and exposing scheduling problems, TAM can serve a useful conceptual tool that allows the system designer to identify and fix *all* the problems in an existing system, or to design a problem-free architecture for a new system. In addition, TAD enables visual analysis, making it clear where problems arise, while the TAM-based simulation can be used to study how scheduling problems actually manifest given certain workloads and resource configurations.

Do the five categories of problems exhaustively describe how system structure could hinder scheduling? For now we can only answer this question empirically. We analyzed systems with distinct architectures (thread-based vs. loose SEDA vs. strict SEDA) and thread behaviors (kernel- vs. user-level threads). Only these problems arise and fixing them allows us to realize various scheduling policies effectively. We leave proving the completeness of the problems to future work.

4 HBase: A Case Study

Given the TAM of a system, multiple scheduling problems may be discovered, pointing towards solutions that introduce necessary scheduling controls. By fixing these problems, the system can be transformed to provide schedulability. We now perform such analysis on a realistic storage system, the HBase/HDFS storage stack (hereinafter just HBase). We focus on

HBase, as it presents the most complex architecture, is widely deployed in many production environments [24], and achieving schedulability remains difficult despite repeated attempts [3, 5, 37, 63, 68]. We analyze the schedulability of MongoDB [15], Cassandra [36] and Riak [33] later (§5).

4.1 TAM simulations

We simulate an HBase cluster with 8 nodes; one master node hosts the HMaster and NameNode, and 7 slave nodes host RegionServers and DataNodes. Each node has a 1 GHz CPU, 100 MB/s disk, and a 1 Gbps network connection. Using this simulation, we compare the orignal HBase (Orig-HBase) and the HBase modified with our solutions (Tamed-HBase, with its TAD shown in Figure 14); later we implement the solutions to show that our TAM-based simulation corresponds to the real world. The solutions can be used to realize any scheduling policy; in our simulation the schedulers simply attempt to isolate C1's performance from C2's workload changes.

4.1.1 No Scheduling

Problem: The Data Xceive and Data Stream stages in HBase have a non-empty resource vector and `on_demand` scheduling type, indicating the no scheduling problem.
Solution: In the Tamed-HBase TAM, we change the scheduling type of Data Xceive and Data Stream from `on_demand` to `pluggable`, so it is free from no scheduling. In a real system, this corresponds to adding scheduling points to the two stages and exporting an API to allow different schedulers to be plugged into each.

We simulate a workload where C1 and C2 keep issuing (uncached) Gets, each of which incurs 128 K I/O at Data Xceive. C1 has 40 threads issuing requests in parallel; the number of threads of C2 increases from 40 to 200. Figure 13(a) shows that even though the original TAM does not isolate C1 from C2, our modified TAM provides stable throughput to C1 despite the change of C2.

4.1.2 Unknown Resource Usage

Problem: In HBase TAM, the I/O and network components of the RPC Handle resource vector take the `unknown` value, indicating unknown resource usage.

Further code inspection reveals that the RPC Handle threads only sends responses when the network is idle, so it does not interfere with scheduling. TAM produces a false positive here because the threads exhibited "scheduler-like" behavior (deciding whether to perform a task based on the status of the resource) without going through the schedulers, which is not captured by TAM. Short-circuited reads, which are unknown when the request is scheduled, do cause contention for I/O and lead to ineffective scheduling.
Solution: We remove the unknown resource usage in the RPC Handle stage by moving short-circuited reads from RPC Handle to Data Xceive. Instead of performing reads by itself, once the RPC Handle stage recognizes a short-circuited read, it directly passes the read to the local Data Xceive stage without going through network transferring; at this point, the Data Xceive scheduler has knowledge of the I/O size and locations.

We simulate a standalone HBase node, which ensures that all HDFS reads at the RegionServer are short-circuited, thus isolating the effect of unknown resource usage. In Figure 13(b), both C1 and C2 issue Gets on cold data, which incurs 100 KB short-circuited reads at RPC Handle. C2 also issues cached Gets that do not require I/O. One can see that Tamed-HBase achieves additional throughput for the cached Gets of C2 compared to Orig-HBase, without reducing the throughput of C1 or C2's cold-cached Gets.

4.1.3 Hidden Contention

Problem: Within the same node of the HBase TAM, both the RPC Handle and Data Xceive stages have an I/O component in their resource vectors; the RPC Read, RPC Handle, RPC Respond, Data Stream, and Data Xceive stage resource vectors all share the network component; many stage resource vectors contain the CPU component. All of them lead to the hidden contention problem.
Solution: To remove hidden contention, we restructure the stages so that in Tamed-HBase, each resource is managed by one dedicated stage. In general, one cannot completely eliminate hidden contention by dividing stages based on resource usage for two reasons:

1. Without special hardware, network packet processing requires significant CPU [16, 20, 30], so the network stage inevitably incurs both network and CPU usage.
2. Lock usage typically cannot be separated to a dedicated stage: it may be pointless to obtain a lock without doing some processing and consuming other resources.

In the case of HBase, the highly contended namespace lock is obtained to perform namespace manipulation (not shown in the simplified TAD), which does not incur extensive usage on other resources, so the lock stage can be separated. The network stage in Tamed-HBase does incur CPU usage; however, by moving most CPU intensive tasks to the CPU stage (e.g., serialization and checksum verification), we can reduce the hidden contention on CPU between the network stage and the CPU stage to a minimal level. The restructured stages are shown in Figure 14; to avoid no scheduling, all the new stages have `pluggable` scheduling points, but the blocking relationships and order constraints are inherited from the old stages to the new ones (until further fixes).

We simulate a workload where C1 and C2 keep issuing 1 KB RPC requests. C1's response size remains 20 KB, while C2's response size varies from 10 to 200 KB. Figure 13(c) shows Tamed-HBase, with the hidden con-

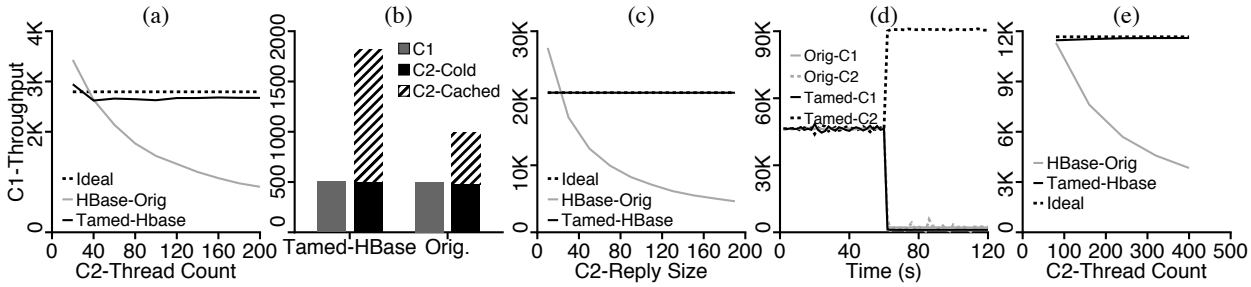


Figure 13: **Tamed-HBase Simulation.** *Problem and solution for (a) no scheduling; (b) unknown resource usage; (c) hidden contention; (d) blocking; (e) ordering constraints.*

tention on network removed, isolates C1 from C2's reply size change; Orig-HBase cannot provide isolation.

4.1.4 Blocking

Problem: In the HBase TAM, three stages are bounded and have a non-empty blocking set: RPC Handle, Mem Flush, and Log Sync, suggesting the blocking problem (which actually occurs in production [5]).

Solution: In Tamed-HBase (with stages restructured to remove hidden contention), we make the CPU and Log Sync stage asynchronous to fix the blocking problem.

In Figure 13(d) we simulate a workload where initially both C1 and C2 issue cached Gets. At time 60s C2 request uncached data, causing threads to block on I/O. When C2 switches to an I/O intensive workload, Tamed-HBase allows C1 to achieve high throughput. In contrast, Orig-HBase delivers very low throughput even though the system has enough resources to process C1 requests.

4.1.5 Ordering Constraints

Problem: In the HBase TAM, the Data Stream and Data Xceive stage have ordering constraints and resource usages, which points to the ordering constraint problem (the Log Sync stage also has ordering constraint, but does not incur extensive usage on any resources, so does not lead to the ordering constraint problem).

Solution: By re-designing the consistency mechanism, the ordering constraint can be removed. For example, each client can maintain a separate WAL, thus eliminating the need to preserve request ordering across clients and removing the ordering constraint in the Log Append and I/O stage in Tamed-HBase.

We simulate a workload where both C1 and C2 issue 1 KB Puts, resulting in 1 KB WAL appends. Figure 13(e) shows that unlike in Orig-HBase, where the throughput drops sharply as C2 issues more requests, Tamed-HBase, with the ordering constraint removed, is able to isolate C2's effect on C1.

4.1.6 Discussion

HBase does attempt to provide scheduling, in the form of exporting a scheduling API at the RPC Handle stage; however, this effort is rather incomplete as it fails to solve any of the scheduling problems HBase possesses, thus

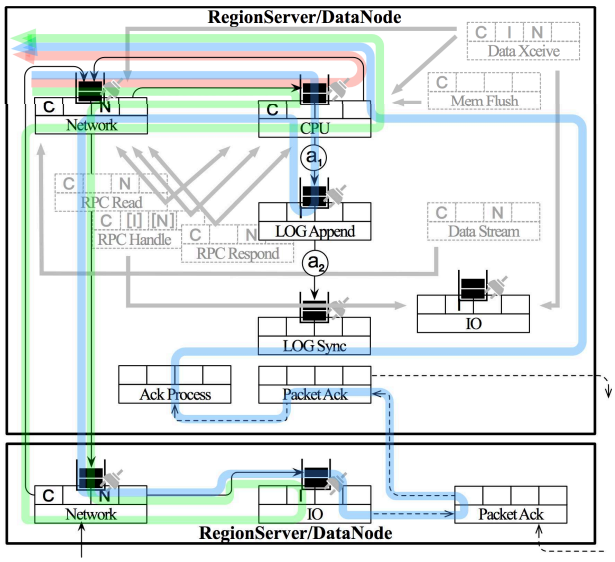


Figure 14: **Tamed-HBase Thread Architecture.** *Stages in grey are replaced by new stages.*

suggesting the importance of systematic schedulability analysis. The TAD of Tamed-HBase is shown in Figure 14. With the aid of TAM, we are able to identify and solve all of HBase's scheduling problems (except for the hidden contention on CPU, which we reduce to a low level), and transform HBase to provide schedulability.

4.2 Implementing Schedulable HBase

In this section, we demonstrate that real HBase suffers from the scheduling problems we identified, and fixing these problems leads to schedulability. The schedulable HBase implementation gives us experience realizing schedulability in real systems and validates that the TAM simulations are excellent predictors of the real world.

To match the simulation environment, we run experiments on an 8-node cluster. Each node has two 8-core CPUs at 2.40 GHz (plus hyper-threading), 128 GB of RAM, an 480 GB SSD (to run the system) and two 1.2 TB HDD (to host the HDFS data). The nodes are connected via 10 Gbps network. One node hosts the HMaster, NameNode, and Secondary NameNode; the other seven nodes host RegionServers and DataNodes.

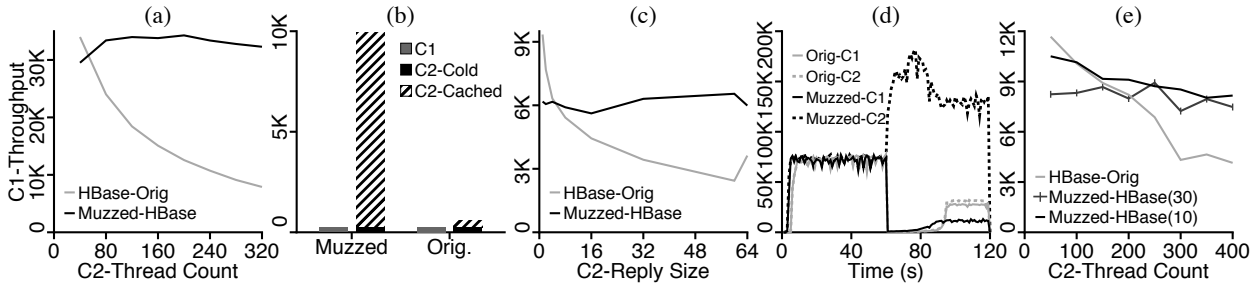


Figure 15: **Muzzled-HBase Implementation.** *The sub-figures repeat Figure 13(a)-(e), respectively (the axis scales are different). To isolate the effect of short-circuit reads, in (b) a standalone node is used instead of a cluster. To ensure that the RegionServer network bandwidth is the bottleneck, in (c) the bandwidth is limited to 100 Mpbs using dummynet [49].*

4.2.1 Schedulability in Real Implementation

In §4.1 we showed via simulation how Tamed-HBase solves the scheduling problems by changing the thread architecture. However, realizing these changes in implementation may be too difficult and risky; for example, making stages asynchronous requires changing the RPC programming model of HBase, and removing the ordering constraint is akin to re-designing the consistency mechanism. Thus in implementation we use various approaches to alleviate the effects of the scheduling problems and approximate the control achieved by Tamed-HBase, with minimal changes to the TAM; we call the resulted implementation Muzzled-HBase (for it is not completely tamed).

No Scheduling: We add scheduling points to the Data Xceive and Data Stream stages to fix the no scheduling problem in HBase. Figure 15(a) illustrates that HBase suffered the no scheduling problem and as a result, the throughput of client C1 is significantly harmed when C2 issues more requests; further, it shows that adding scheduling points at resource-intensive stages provides performance isolation in the real-world.

Unknown Resource Usage: For ease of implementation, in Muzzled-HBase we do not move short-circuited read processing to Data Xceive, as we did for Tamed-HBase (§4.1.2). Instead, we keep the TAM unchanged and use speculative execution to work around this problem. We track the workload pattern of each client; when the CPU is idle, we speculatively execute requests from the CPU-intensive clients. If, during speculation, a request is found to require I/O, it is aborted and put back on the queue where it is subjected to normal scheduling.

The unknown resource problem that exists in HBase is shown in Figure 15(b): when client C2 requests in-cache data, Orig-HBase is not able to efficiently utilize the CPU. Muzzled-HBase with speculative execution dramatically improves the throughput of C2 without harming C1, achieving roughly the same effects as Tamed-HBase, though at the cost of wasted CPU cycles.

Hidden Contention: The complete solution to the hidden contention problem requires restructuring the TAM; this is further complicated by the fact that these stages reside in two separate processes (RegionServer and DataNode). For implementation simplicity, in Muzzled-HBase we only combine the RPC Read and RPC Respond stage, which are mostly responsible for network resource consumption. We work around the remaining contention by having a controller monitor resource usage and adjust client weights at each stage. If stage S1 is excessively using resource on behalf of client C1, the weight of C1 is reduced across *all* stages so that fewer C1 requests are issued to S1, forcing S1 to either use fewer resources or serve other clients; this algorithm is similar to the one deployed in Retro [37].

Figure 15(c) verifies that HBase suffered hidden contention across multiple stages, which manifests when one stage consumes more resources on behalf of a particular client (i.e., more network for C2). The small difference between the implementation and simulation results for a reply size of 64KB occurs because in the implementation, after transferring 64KB, the RPC Respond thread switches to another request; we did not simulate this detail. With two network-intensive stages combined and cross-stage coordination, Muzzled-HBase is able to control the hidden contention and largely ensures isolation, though incurs extra communication overheads.

Blocking: We work around the blocking problem in the RPC Handle stage in HBase without changing the RPC programming model by treating RPC Handle threads as a resource and allocating them between clients like CPU or I/O resources. This approach does not eliminate blocking, but prevents one client from occupying all threads and allows other clients to make progress.

The blocking problem that exists within HBase is illustrated in Figure 15(d). In Orig-HBase, when the workload of one client switches from CPU to I/O-intensive (C2 at time 60), both clients are harmed because not enough threads are available. Our solution, however, protects C1 from the workload change of C2. The slight difference in the implementation and simulation results occurs because we did not simulate page cache effects.

Ordering Constraint: Directly removing ordering constraints from the TAM would require re-designing the consistency mechanism of HBase. In Muzzled-HBase,

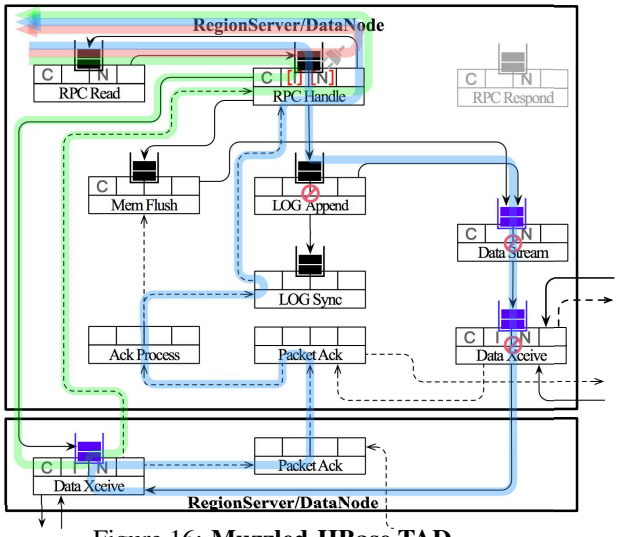


Figure 16: **Muzzled-HBase TAD.**

we work around this by scheduling at RPC Handle, above the ordering-constrained LOG Append stage. Note that we already schedule based on RPC Handle time in this stage to solve the blocking problem. Since threads block until WAL writes are done, under a stable workload, blocking time is roughly proportional to the number of downstream requests, and scheduling RPC Handle time indirectly schedules the WAL writes before passing it to LOG Append. However, the number of RPC Handle threads are typically larger than the I/O parallelism in the system, making this approach less effective; therefore, we compare two settings of Muzzled-HBase, with 10 or 30 RPC Handle threads.

HBase's ordering problem is shown in Figure 15(e); when C2 writes more data, the throughput of C1 suffers. Again, this problem is alleviated in Muzzled-HBase by limiting the number of outstanding requests to the lower stage to 10 or 30; 30 outstanding requests leads to worse isolation than 10, as C1 competes with more requests from C2 after they enter RPC Handle.

Summary: The TAD of Muzzled-HBase is shown in Figure 16. We can see that the no scheduling problem and the hidden contention between RPC Read and RPC Respond are fixed. However, it still exhibits other problems, including unknown resource usage, blocking, ordering constraint, and hidden contention among other stages; changing the thread architecture of HBase to fix these problems would be too difficult. Various approaches are used instead to mitigate the effects of these problems and achieve approximate scheduling control, but these approaches also incur overheads (e.g., wasted CPU cycles on aborted requests or communication across stages).

On top of Muzzled-HBase, multiple scheduling policies are implemented, including FIFO, DRF and priority scheduling. Client identifiers are propagated across stages with requests, so each scheduler can map requests back to the originating client. In our implementation, a centralized controller collects information and coordinates local scheduler behavior; however, other mechanisms such as distributed coordination are also possible. For now the scheduler only performs network resource scheduling on server bandwidth; we anticipate incorporating global network bandwidth allocation [35] in the future. The final implementation consists of ∼4800 lines of code modification to HBase and ∼3000 to HDFS.

The performance of Muzzled-HBase for YCSB [17] is shown in Figure 1. For Figure 1(a), five clients are each given a different weight and we use DRF-based local scheduler to achieve global weighted fairness. Orig-HBase was unable to provide weighted fairness across clients with different priorities, instead delivering approximately equal throughput to each; Muzzled-HBase, in contrast, enforces weighted fairness as desired. For Figure 1(b), priority scheduling is implemented atop Muzzled-HBase by always reserving a small subset of resources, including the RPC Handle threads for foreground workloads. With Orig-HBase, the tail latencies of the foreground workload increase significantly when different types of workloads run in background; Muzzled-HBase, however, is able to maintain stable latencies despite the interference from the background workloads.

4.2.2 Discussion

Schedulability can be achieved by modifying the problematic TAM to eliminate scheduling problems. However, as we can see in the case of HBase, changing the TAM for existing systems usually involves restructuring the system, which is labor-intensive. To minimize the changes to the architecture or lower the engineering effort, often we are forced to keep the same TAM, but use various approaches to work around its inherent structural flaws and alleviate the effects of the scheduling problems. Unfortunately, these approaches only provide approximate scheduling control and incur overheads.

We thus encourage developers to take schedulability into consideration in the early phase of system design; this is especially important in a cloud-based world where users demand isolation and quality of service guarantees. By specifying the TAM of a system, potential scheduling problems can be discovered early, avoiding the painful process of retrofitting scheduling control later. Of course, schedulability may need to be balanced with other system design goals. For example, the system architects may decide that having a simple synchronous programming model is more important, and accept blocking at some stages. However, these kind of compromises should be made only after carefully weighing the trade-offs between different goals, not just due to the obliviousness of their schedulability ramification.

5 Schedulability of Other Systems

Earlier we showed how to transform HBase to provide schedulability. Other concurrent systems can be analyzed and tranformed in the same way. Here, we analyze the schedulability of MongoDB [15], Cassandra [36], and Riak [33]. Table 2 presents a summary of their scheduling problems. Some of the problems predicted by TAM have been experienced in production environments [4, 5, 6]; these problems and their solutions have also been verified by simulation results (see [67]).

MongoDB: The TAD of MongoDB is shown in Figure 3. From its TAM we can identify (a) the unknown resource usage problem at the Worker stage, which processes client requests until completion; (b) the hidden contention problem in the secondary node; most notably, the Worker and Oplog Writer stages compete for database locks, causing reads to have unbounded delay under heavy write load, which is reported in production [6]; (c) the blocking problem at the Worker stage.

Lessons: MongoDB resembles the traditional thread-per-request architecture and thus suffers unknown resource usage, which stems from the complex execution path within one thread. The complex path and resource patterns within the Worker stage makes it challenging to work around this problem. We expect that altering MongoDB to provide schedulability will be difficult and may require substantial structural changes.

Cassandra: The TAD of Cassandra is shown in Figure 4. From its TAM we identify (a) unknown resource usage in the Read, Mutation, View-Mutation stages since those stages may perform I/O; (b) hidden contention between many stages for CPU, I/O and network; (c) blocking in the C-ReqHandle stage.

Lessons: Cassandra closely follows the standard SEDA architecture, where all activities are managed in controlled stages; unfortunately, schedulability does not automatically follow. Too many stages with the same resource pattern leads to hidden contention and the "inability to balance reads/writes/compaction/flushing", as reported by developers [4]; likewise, CPU- and I/O-intensive operations in the same stage leads to unknown resource usage. More thoughts on how to divide stages are needed to build a highly schedulable system. Instead of dividing stages based on functionality, we recommend dividing stages based on resource usage patterns to give more resource information to the scheduler and reduce hidden competition. Cassandra is currently moving toward this direction: developers have proposed combining different processing stages into a single non-blocking stage, and moving I/O to a dedicated thread pool [4].

Riak: The TAD of Riak is shown in Figure 5. From its TAM we can identify (a) the no scheduling problem at the Req In-Out and Req Process stages; (b) unknown resource usage in the Process and Cmd Handle stages; (c) hidden contention across all stages.

	N	U	C	B	O
HBase [24]	✖	✖	✖	✖	✖
MongoDB [15]		✖	✖	✖	
Cassandra [36]		✖	✖	✖	
Riak [33]	✖	✖	✖		

Table 2: **Scheduling Problems Identified From TAM.** ✖*:have the corresponding problem.*

Lessons: Riak also closely follows the SEDA architecture. Riak relies heavily on light-weighted processes and transparent IPC provided by the Erlang virtual machine, which makes resource management implicit [22]. Creating a new Erlang process may have low overhead; creating them on-demand leads to the no scheduling problem. Similarly, with transparent IPC, many stages may consume network bandwidth without knowing it, causing unknown resource usage and hidden contention. To make Riak schedulable, one must either explicitly manage the above mechanisms, or change Erlang VM to allow scheduling policies to be passed from Riak to the VM, which manges the resources.

6 Model Limitations

We have shown that TAM is a useful tool for schedulability analysis and delivers promising results. In this section we discuss some of its limitations and how we can extend TAM to further help schedulability analysis.

First, current TAM is best suited for describing SEDA-like systems, where each thread belongs to a specific stage. However, in other concurrency models, threads and stages may not be statically bound. For example, in a run-to-completion model, a single thread may perform multiple tasks until a request is completed, and be scheduled (possibly by yielding) before each task. In this case, a stage would be better defined as the execution between scheduling points, allowing one thread to cross multiple stages. We leave extending TAM to other concurrency models to future work.

Second, various workarounds can be used to mitigate the effects of scheduling problems; most of them involve coordination among stages or predicting workload characteristics. Encoding these mechanisms into TAM, possibly in the form of information flow between stages, would allow it to capture the scheduling effects of indirect workarounds.

Finally, even though different systems might possess the same scheduling problems, the difficulty of fixing their problems could vary vastly based on the system's internal structure and code base. Fixing the unknown resource problem directly in HBase requires only separating the short-circuited read processing from the RPC Read stage; fixing this problem in MongoDB, however, requires a major re-structuring of the Worker stage to account for its complex execution paths. TAM is effective

in identifying the problems, but does not give many indications on how difficult solving these problems would be; systematically reasoning about such difficulties is another interesting direction to extend TAM.

7 Related Work

Scheduling as a general problem has been extensively studied in computer science, manufacturing, operational research, and many other fields [34, 52, 53, 60]. Our work differs from the previous ones as we separate the scheduling problem in distributed storage systems into two sub-problems: the meta *schedulability* problem and the specific scheduling problem. For a general-purpose storage system that is designed to work for various workloads and meet various peformance measures, the schedulability problem is answered at the system design/build phase, and concerns whether the system offers proper *scheduling support*: are schedulers placed at the right points in the system and given necessary information and control? Once proper scheduling support is built in (i.e., the system provides schedulability), the user can solve his/her own specific scheduling problem: given her workload, which scheduling policy should she implement on top of the scheduling support provided by the system to realize a particular performance goal?

Such separation distinguishes the TAM approach from other formalization of the scheduling problems, such as queuing networks [13, 32, 41, 59] or stochastic Petri nets [18, 46, 60, 70, 71], which focus on solving specific scheduling problems. For example, traditional queuing network models encode specific scheduling plan information and workload characteristics, and output performance measures [43, 50, 59]. One could view TAM as a queuing network skeleton, stripped of all information but that available at system design time; our schedulability analysis aims to derive properties from the limited information encoded in TAM that would hold after the TAM skeleton is augmented with various workload/queing discipline information to form a complete queuing network. Some techniques developed in the queuing theory context may be borrowed to prove certain properties of the TAM [31, 65]; we leave that as future work.

From a more system-oriented perspective, previous work has focused on proposing scheduling plans that achieve various specific goals [47, 54, 55, 57, 63, 68]. For example, Pisces [55] discusses how to allocate local weights to match client demands and achieve global fairness; Cake [63] proposes a feedback loop to adjust local scheduler behavior to provide latency guarantees; Retro [37] supports different scheduling policies, but by translating these policies into rate limits at local schedulers. All the above works need proper scheduling support to enforce their plans. As current systems usually lack such support (§5), people indeed encouter the five categories of problems we have identified during the realization of their scheduling plans [38, 54, 63, 68]: Mace et al. found that unknown resource usage and blocking prevented them from achieving fairness [38]; Cake [63] had to add scheduling points to HDFS to enforce SLOs. However, in these systems the encountered problems are solved in an ad hoc manner; the solutions are often buried in implementation details or not discussed at all. A general framework that addresses the schedulability problem explicitly and systematically is thus strongly called for.

Monotasks [44] advocates an architecture in which jobs are broken into units of work that each use a single resource, and each resource is managed with a dedicated scheduler. From the TAM perspective, such an architecture eliminates the unknown resource usage and the hidden contention problem, allowing the system to provide better schedulability. The authors indeed observe that this architecture "allows MonoSpark to avoid resource contention and under utilization", as predicted by TAM.

Our work is also similar to SEDA [64] and Flash [45] in the sense that it studies and modifies the thread structure and interactions to improve system performance. Like our work, Capriccio [61] automatically deduces a flow graph and places scheduling points at the graph nodes for thread scheduling.

8 Conclusions

With sharing being one of the key aspects of modern scalable storage systems, correct and flexible scheduling becomes a central goal in system design. To ensure scheduling works as desired, schedulability analysis should be included as an integrated part of the concurrency architecture. The thread architecture model provides a systematic way of performing such analysis, thus turning the art of enabling effective scheduling into a science that is easily accessible and automatable. The software for schedulability analysis (e.g., TADalyzer and the TAM-based simulation framework) is available at http://research.cs.wisc.edu/adsl/Software/TAM.

Acknowledgements

We thank Rebecca Isaacs (our shepherd), the members of ADSL, and the anonymous reviewers for their feedback. We are especially grateful to the reviewer who asked for the model behind the thread architecture diagrams. We thank Michael Gleicher for his guidance in the thread architecture visualization. CloudLab [48] provided infrastructure to run our experiments. This material was supported by funding from NSF grants CNS-1421033, CNS-1218405, and CNS-1838733, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, NetApp, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[2] Byteman Homepage. http://byteman.jboss.org/.

[3] HBase Issue Tracking Page. https://issues.apache.org/jira/browse/HBASE-8884, July 2013.

[4] Cassandra Issues: Move away from SEDA to TPC. https://issues.apache.org/jira/browse/CASSANDRA-10989, January 2016.

[5] HBase Issue Tracking Page. https://issues.apache.org/jira/browse/HBASE-8836, July 2016.

[6] MongoDB Issue Tracking Page. https://jira.mongodb.org/browse/SERVER-24661, July 2016.

[7] MongoDB Issue Tracking Page. https://jira.mongodb.org/browse/SERVER-20328, July 2016.

[8] ANANDKUMAR, A., BISDIKIAN, C., AND AGRAWAL, D. Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints. In *ACM SIGMETRICS Performance Evaluation Review* (2008), vol. 36, ACM, pp. 133–144.

[9] ANGEL, S., BALLANI, H., KARAGIANNIS, T., O'SHEA, G., AND THERESKA, E. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI* (2014), pp. 233–248.

[10] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[11] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18.

[12] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)* (Vancouver, Canada, December 2010).

[13] BOXMA, O. J., KOOLE, G. M., AND LIU, Z. *Queueing-Theoretic Solution Methods for Models of Parallel and Distributed Systems*. Centrum voor Wiskunde en Informatica, Department of Operations Research, Statistics, and System Theory, 1994.

[14] CHEN, Y.-Y. M., ACCARDI, A. J., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. Path-Based Failure and Evolution Management.

[15] CHODOROW, K. *MongoDB: the Definitive Guide*. " O'Reilly Media, Inc.", 2013.

[16] CLARK, D. D., JACOBSON, V., ROMKEY, J., AND SALWEN, H. An analysis of TCP processing overhead. *IEEE Communications magazine 27*, 6 (1989), 23–29.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[18] DAVIDRAJUH, R. Activity-Oriented Petri Net for scheduling of resources. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on* (2012), IEEE, pp. 1201–1206.

[19] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[20] DIWAKER GUPTA, L. C., GARDNER, R., AND VAHDAT, A. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)* (Melbourne, Australia, Nov 2006).

[21] ELSER, J. Personal Correspondence, April 2017.

[22] FINK, B. Distributed Computation on Dynamo-Style Distributed Storage: Riak Pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop* (2012), ACM, pp. 43–50.

[23] FINK, B. Personal Correspondence, April 2017.

[24] GEORGE, L. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. " O'Reilly Media, Inc.", 2011.

[25] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003), pp. 29–43.

[26] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI* (2011), vol. 11, pp. 24–24.

[27] GU, D. Personal Correspondence, April 2017.

[28] GULATI, A., AHMAD, I., WALDSPURGER, C. A., ET AL. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST* (2009), vol. 9, pp. 85–98.

[29] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 13–24.

[30] KAY, J., AND PASQUALE, J. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *ACM SIGCOMM Computer Communication Review* (1993), vol. 23, ACM, pp. 259–268.

[31] KINGMAN, J. The heavy traffic approximation in the theory of queues. In *Proceedings of the Symposium on Congestion Theory* (1965), no. 2, University of North Carolina Press, Chapel Hill, NC.

[32] KLEINROCK, L. *Queueing Systems, Volume 2: Computer Applications*, vol. 66. wiley New York, 1976.

[33] KLOPHAUS, R. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming* (2010), ACM, p. 14.

[34] KOLISCH, R., AND SPRECHER, A. PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research 96*, 1 (1997), 205–216.

[35] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ET AL. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 1–14.

[36] LAKSHMAN, A., AND MALIK, P. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (Big Sky Resort, Montana, Oct 2009).

[37] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *NSDI* (2015), pp. 589–603.

[38] MACE, J., BODIK, P., MUSUVATHI, M., FONSECA, R., AND VARADARAJAN, K. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 144–159.

[39] MATLOFF, N. Introduction to Discrete-Event Simulation and the SimPy Language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August 2* (2008), 2009.

[40] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter* (1993), vol. 93.

[41] MENGA, G., BRUNO, G., CONTERNO, R., AND DATO, M. Modeling FMS by closed queuing network analysis methods. *IEEE transactions on components, hybrids, and manufacturing technology 7*, 3 (1984), 241–248.

[42] MUTHUKKARUPPAN, K. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)* (Pacific Grove, California, October 2011).

[43] NGUYEN, H., SHEN, Z., GU, X., SUBBIAH, S., AND WILKES, J. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *ICAC* (2013), vol. 13, pp. 69–82.

[44] OUSTERHOUT, K., CANEL, C., RATNASAMY, S., AND SHENKER, S. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 184–200.

[45] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference, General Track* (1999), pp. 199–212.

[46] PETERSON, J. L. Petri Nets. *ACM Comput. Surv. 9*, 3 (September 1977), 223–252.

[47] REDA, W., CANINI, M., SURESH, L., KOSTIĆ, D., AND BRAITHWAITE, S. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 95–110.

[48] RICCI, R., EIDE, E., AND TEAM, C. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE 39*, 6 (2014), 36–38.

[49] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[50] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open Versus Closed: A Cautionary Tale.

[51] SERENYI, D. Cluster-Level Storage at Google. www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf, 2017.

[52] SHIH, H. M., AND SEKIGUCHI, T. A timed Petri net and beam search based online FMS scheduling system with routing flexibility. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on* (1991), IEEE, pp. 2548–2553.

[53] SHIRAZI, B. A., KAVI, K. M., AND HURSON, A. R. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[54] SHUE, D., AND FREEDMAN, M. J. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 17.

[55] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI* (2012), vol. 12, pp. 349–362.

[56] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)* (Incline Village, Nevada, May 2010).

[57] SURESH PUTHALATH, L. On predictable performance for distributed systems.

[58] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 182–196.

[59] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An Analytical Model for Multi-Tier Internet Services and Its Applications. In *ACM SIGMETRICS Performance Evaluation Review* (2005), vol. 33, ACM, pp. 291–302.

[60] VAN DER AALST, W. M. The Application of Petri Nets to Workflow Management. *Journal of circuits, systems, and computers 8*, 01 (1998), 21–66.

[61] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003), pp. 268–281.

[62] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)* (San Jose, California, February 2007).

[63] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., KATZ, R., AND STOICA, I. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), ACM, p. 14.

[64] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Banff, Canada, October 2001).

[65] WHITT, W. A Light-Traffic Approximation for Single-Class Departure Processes from Multi-Class Queues. *Management science 34*, 11 (1988), 1333–1346.

[66] XU, Y., AND ZHAO, M. IBIS: Interposed Big-Data I/O Scheduler. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (2016), ACM, pp. 111–122.

[67] YANG, S. *Schedulability in Local and Distributed Storage Systems*. The University of Wisconsin-Madison, 2017.

[68] ZENG, J., AND PLALE, B. Workload-Aware Resource Reservation for Multi-tenant NoSQL. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on* (2015), IEEE, pp. 32–41.

[69] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *OSDI* (2014), vol. 14, pp. 629–644.

[70] ZHOU, M., AND DICESARE, F. Parallel and Sequential Mutual Exclusions for Petri Net Modeling of Manufacturing Systems with Shared Resources. *IEEE Transactions on Robotics and Automation 7*, 4 (1991), 515–527.

[71] ZHOU, M., AND WU, N. *System Modeling and Control with Resource-Oriented Petri Nets*, vol. 35. Crc Press, 2009.

[72] ZHU, T., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.

μTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman Thomas F. Wenisch

University of Michigan
akshitha@umich.edu, twenisch@umich.edu

ABSTRACT

Modern On-Line Data Intensive (OLDI) applications have evolved from monolithic systems to instead comprise numerous, distributed microservices interacting via Remote Procedure Calls (RPCs). Microservices face sub-millisecond (sub-ms) RPC latency goals, much tighter than their monolithic counterparts that must meet $\geq$ 100 ms latency targets. Sub-ms–scale threading and concurrency design effects that were once insignificant for such monolithic services can now come to dominate in the sub-ms–scale microservice regime. We investigate how threading design critically impacts microservice tail latency by developing a *taxonomy of threading models*—a structured understanding of the implications of how microservices manage concurrency and interact with RPC interfaces under wide-ranging loads. We develop μ*Tune*, a system that has two features: (1) a novel framework that abstracts threading model implementation from application code, and (2) an automatic load adaptation system that curtails microservice tail latency by exploiting inherent latency trade-offs revealed in our taxonomy to transition among threading models. We study μ*Tune* in the context of four OLDI applications to demonstrate up to 1.9× tail latency improvement over static threading choices and state-of-the-art adaptation techniques.

1 Introduction

On-Line Data Intensive (OLDI) applications, such as web search, advertising, and online retail, form a major fraction of data center applications [113]. Meeting soft real-time deadlines in the form of Service Level Objectives (SLOs) determines end-user experience [21, 46, 55, 95] and is of paramount importance. Whereas OLDI applications once had largely monolithic software architectures [50], modern OLDI applications comprise numerous, distributed microservices [66, 90, 116] like HTTP connection termination, key-value serving [72], query rewriting [48], click tracking, access-control management, protocol routing [25], etc. Several companies, such as Amazon [6], Netflix [1], Gilt [37], LinkedIn [17], and SoundCloud [9], have adopted microservice architectures to improve OLDI development and scalability [144]. These microservices are composed via standardized Remote Procedure Call (RPC) interfaces, such as Google's Stubby and `gRPC` [18] or Facebook/Apache's Thrift [14].

Whereas monolithic applications face $\geq$ 100 ms tail (99^{th}+%) latency SLOs (e.g.,$\sim$300 ms for web search [126, 133, 142, 150]), microservices must often achieve sub-ms (e.g., $\sim$100 μs for protocol routing [151]) tail latencies as many microservices must be invoked serially to serve a user's query. For example, a Facebook news feed service [79] query may flow through a serial pipeline of many microservices, such as (1) `Sigma` [15]: a spam filter, (2) `McRouter` [118]: a protocol router, (3) `Tao` [56]: a distributed social graph data store, (4) `MyRocks` [29]: a user database, etc., thereby placing tight sub-ms latency SLOs on individual microservices. We expect continued growth in OLDI data sets and applications to require composition of ever more microservices with increasingly complex interactions. Hence, the pressure for better microservice latency SLOs continually mounts.

Threading and concurrency design have been shown to critically affect OLDI response latency [76, 148]. But, prior works [71] focus on monolithic services, which typically have $\geq$ 100 ms tail SLOs [111]. Hence, sub-ms–scale OS and network overheads (e.g., a context switch cost of 5-20 μs [101, 141]) are often insignificant for monolithic services. However, sub-ms–scale microservices differ intrinsically: spurious context switches, network/RPC protocol delays, inept thread wakeups, or lock contention can dominate microservice latency distributions [39]. For example, even a single 20μs spurious context switch implies a 20% latency penalty for a request to a 100 μs SLO protocol routing microservice [151]. Hence, prior conclusions must be revisited for the microservice regime [49].

In this paper, we study how threading design affects mi-

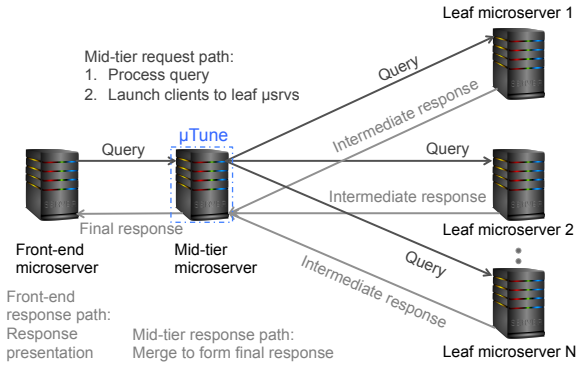


Figure 1: A typical OLDI application fan-out.

croservice tail latency, and leverage these design effects to dynamically improve tails. We develop a system called *µTune*, which features a framework that builds upon open-source RPC platforms [18] to enable microservices to abstract threading model design from service code. We analyze a *taxonomy of threading models* enabled by *µTune*. We examine synchronous or asynchronous RPCs, in-line or dispatched RPC handlers, and interrupt- or poll-based network reception. We also vary thread pool sizes dedicated to various purposes (network polling, RPC handling, response execution). These design axes yield a rich space of microservice architectures that interact with the underlying OS and hardware in starkly varied ways. These threading models often have surprising OS and hardware performance effects including cache locality and pollution, scheduling overheads, and lock contention.

We study *µTune* in the context of four full OLDI services adopted from µSuite [134]. Each service comprises sub-ms microservices that operate on large data sets. We focus our study on *mid-tier microservers*: widely-used [50] microservices that accept service-specific RPC queries, fan them out to leaf microservers that perform relevant computations on their respective data shards, and then return results to be integrated by the mid-tier microserver, as illustrated in Fig. 1. The mid-tier microserver is a particularly interesting object of study since (1) it acts as both an RPC client and an RPC server, (2) it must manage fan-out of a single incoming query to many leaf microservers, and (3) its computation typically takes tens of microseconds, about as long as OS, networking, and RPC overheads.

We investigate threading models for mid-tier microservices. Our results show that the best threading model depends critically on the offered load. For example, at low loads, models that poll for network traffic perform best, as they avoid expensive OS thread wakeups. Conversely, at high loads, models that separate network polling from RPC execution enable higher service capacity and blocking outperforms polling for incoming network traffic as it avoids wasting precious CPU on fruitless poll loops.

We find that the relationship between optimal threading model and service load is complex—one could not expect a developer to pick the best threading model a priori. So, we build an intelligent system that uses offline profiling to automatically adapt to time-varying service load.

µTune's second feature is an adaptation system that determines load via event-based load monitoring and tunes both the threading model (polling vs. blocking network reception; inline vs. dispatched RPC execution) and thread pool sizes in response to load changes. *µTune* improves tail latency by up to 1.9× over static peak load-sustaining threading models and state-of-the-art adaptation techniques, with $< 5\%$ mean latency and instruction overhead. Hence, *µTune* can be used to dynamically curtail sub-ms–scale OS/network overheads that dominate in modern microservices.

In summary, we contribute:

- A *taxonomy of threading models*: A structured understanding of microservice threading models and their implications on performance.
- *µTune's framework* [1] for developing microservices, which supports a wide variety of threading models.
- *µTune's load adaptation system* for tuning threading models and thread pools under varying loads.
- A detailed performance study of OLDI services' key tier built with *µTune*: the mid-tier microserver.

2 Motivation

We motivate the need for a threading taxonomy and adaptation systems that respond rapidly to wide-ranging loads.

Many prior works have studied leaf servers [63, 107, 108, 123, 142, 143], as they are typically most numerous, making them cost-critical. *Mid-tier* servers [68, 98], which manage both incoming and outgoing RPCs to many clients and leaves, perhaps face greater tail latency optimization challenges, but have not been similarly scrutinized. Their network fan-out multiplies underlying software stack interactions. Hence, performance and scalability depend critically on mid-tier threading model design.

Expert developers extensively tune critical OLDI services via trial-and-error or experience-based intuition [84]. Few services can afford such effort; for the rest, we must appeal to software frameworks and automatic adaptation to improve performance. *µTune* aims to empower small teams to develop performant mid-tier microservices that meet latency goals without enormous tuning efforts.

The need for a threading model taxonomy. We develop a structured understanding of rational design options for architecting microservices' OS/network interactions in the form of a *taxonomy of threading models*. We

[1] Available at https://github.com/wenischlab/MicroTune

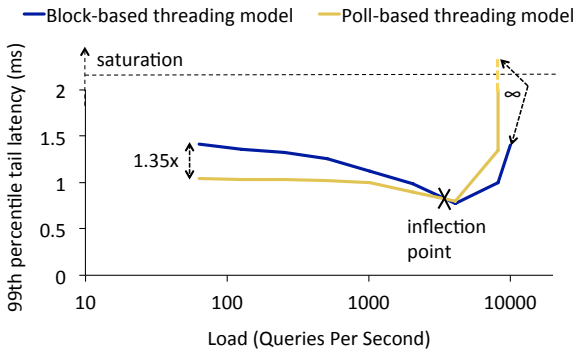


Figure 2: $99^{th}\%$ tail latency for an RPC handled by a block-based & poll-based model: poll-based model improves latency by 1.35x at low load, and saturates at high load.

study these models' latency effects under diverse loads to offer guidance on when certain models perform best.

Prior works [69,83,84,146,148] broadly classify monolithic services as: thread-per-request *synchronous* or event-driven *asynchronous*. We note threading design space dimensions beyond these coarse-grain designs. We build on prior works' insights, such as varying parallelism to reduce tail latency [76], to consider a more diverse taxonomy and spot sub-ms performance concerns.

The need for automatic load adaptation. Subtle changes in a microservice's OS interaction (e.g., how it accepts incoming RPCs) can cause large tail latency differences. For example, Fig. 2 depicts the $99^{th}\%$ tail latency for a sample RPC handled by an example mid-tier microservice as a function of load. We use a mid-tier microserver with 36 physical cores that dispatches requests received from the front-end to a group of worker threads which then invoke synchronous calls to the leaves. The yellow line is the tail latency when we dedicate a thread to poll for incoming network traffic in a CPU-unyielding spin loop. The blue line blocks on the OS socket interface awaiting work to the same RPC handler. We see a stark load-based performance inflection even for these simple designs. At low load, a poll-based model gains 1.35× latency as it avoids OS thread wakeups. Conversely, at high load, fruitless poll loops waste precious CPU that might handle RPCs. The poll-based model becomes saturated, with arrivals exceeding service capacity and unbounded latency growth. Blocking-based models conserve CPU and are more scalable.

We assert that such design trade-offs are not obvious: no single threading model is optimal at all loads, and even expert developers have difficulty making good choices. Moreover, most software adopts a threading model at design time and offers no provision to vary it at runtime.

A microservice framework. Instead, we propose a novel microservice framework in μ*Tune* that abstracts threading design from the RPC handlers. The μ*Tune* system adapts to load by choosing optimal threading models and thread pool sizes dynamically to reduce tail latency.

μ*Tune* aims to allow a microservice to be built once and be scalable across wide-ranging loads. Many OLDI services experience drastic diurnal load variations [79]. Others may face "flash crowds" that cause sudden load spikes (e.g., intense traffic after a major news event). New OLDI services may encounter explosive customer growth that surpasses capacity planning (e.g., the meteoric launch of Pokemon Go [31]). Supporting load scalability over many orders of magnitude in a single framework facilitates rapid scale-up of a popular new service.

3 A Taxonomy of Threading Models

A *threading model* is a software system design choice that governs how responsibility for key application functionality will be divided among threads and how the application will achieve request concurrency. Threading models critically impact the service's throughput, latency, scalability, and programmability. We characterize preemptive instead of co-operative (e.g., node.js [140]) threading models.

3.1 Key dimensions

We identify three threading model dimensions and discuss their programmability and performance implications.

Synchronous vs. asynchronous communication. Prior works have identified synchronous vs. asynchronous communication as a key design choice in monolithic OLDI services [69,83,84,146,148]. Synchronous models map a request to a single thread throughout its lifetime. Request state is implicitly tracked via the thread's PC and stack—programmers simply maintain request state in automatic variables. Threads use blocking I/O to await responses from storage or leaf nodes. In contrast, asynchronous models are event-based—programmers explicitly define state machines for a request's progress [83]. Any ready thread may progress a request upon event reception; threads and requests are not associated.

Programmability: Synchronous models are typically easier to program, as they entail writing straight-forward code without worrying about elusive concurrency-related subtleties. Conversely, asynchronous models require explicit reasoning about request state, synchronization, and races. Ensuing code is often characterized as "spaghetti"—control flow is obscured by callbacks, continuations, futures, promises, and other sophisticated paradigms. Due to this vast programmability gap, we spent three weeks implementing synchronous and four months for asynchronous models.

Performance: As synchronous models await leaf responses before progressing new requests, they face request/response queuing delays, producing worse response latencies and throughput than asynchronous [69,114,146]. Adding more synchronous threads can allay queuing, but

can induce secondary bottlenecks, such as cache pollution, lock contention, and scheduling/thread wakeup delays.

Synchronous apps: Azure SQL [5], Google Cloud SQL's Redmine [10, 100], MongoDB replication [28]

Asynchronous apps: Apache [3], Azure blob storage [27], Redis replication [34], Server-Side Mashup [105], CORBA Model, Aerospike [2]

In-line vs. dispatch-based RPC processing. In in-line models, a single thread manages the entire RPC lifetime, from the point where it is accepted from the RPC library until its response is returned. Dispatch-based models separate responsibilities between network threads, which accept new requests from the underlying RPC interface, and worker threads, which execute RPC handlers.

Programmability: In-line models are simple; thread pools block/poll on the RPC arrival queue and execute an RPC completely before receiving another. Dispatched models are more complex; RPCs are explicitly passed from network to worker threads via thread-safe queues.

Performance: In-line models avoid the explicit state hand-off and thread-hop to pass work from network to worker threads. Hence, they are efficient at low loads and for short requests, where dispatch overheads dominate service times. But, if a single thread cannot sustain the service load, multiple threads contending to accept work typically outweighs hand-off costs, which can be carefully honed. In-line models are prone to high queuing, as each thread processes whichever request it receives. In contrast, dispatched models can explicitly prioritize requests.

In-line apps: Redis [41, 58], MapReduce workers [64]

Apps that dispatch: IBM's WebSphere for z/OS [22, 81], Oracle's EDT image search [20], Mule ESB [12], Malwarebytes [19], Celery for RabbitMQ and Redis [11], Resque [35] and RQ [36] Redis queues, NetCDF [74]

Block- vs. poll-based RPC reception. While the synchronous and in-line dimensions address outgoing RPCs, the block vs. poll dimension concerns incoming RPCs. In block-based models, threads await new work via blocking system calls, yielding CPU if no work is available. Threads block on I/O interfaces (e.g., `read()` or `epoll()` system calls) awaiting work. In poll-based models, a thread spins in a loop, continuously looking for new work.

Performance: The poll vs. block trade-off is intrinsic: polling reduces latency, while blocking frees a waiting CPU to perform other work. Polling incurs lower latency as it avoids OS thread wakeups [106] to which blocking is prone. But, polling wastes CPU time in fruitless poll loops, especially at low loads. Yet, many latency-sensitive services opt to poll [34], perhaps solely to avoid unexpected hardware or OS actions, such as a slow transition to a low-power mode [51]. Many polling threads can contend to cause pathologically poor performance [88].

Apps that block: Redis BLPOP [7]

Apps that poll: Intel's DPDK Poll Driver [32], Redis replication [34], Redis LPOP [24], DoS attacks and defenses [117, 125, 132], GCP Health Checker [38]

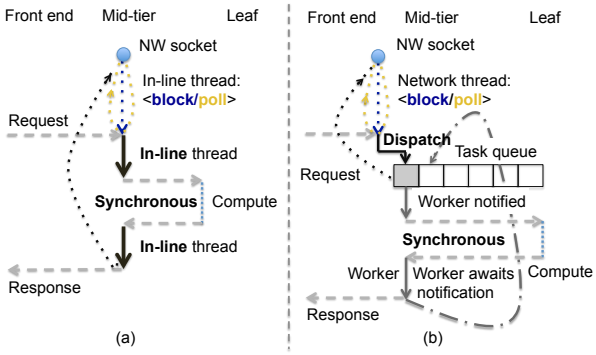


Figure 3: Execution of an RPC by (a) SIB/SIP (b) SDB/SDP

These three dimensions lead to eight mid-tier threading models. We also vary thread pool sizes for these models.

3.2 Synchronous models

In synchronous models, we create maximally sized thread pools on start-up and then "park" extraneous threads on condition variables, to rapidly supply threads as needed without `pthread_create()` call overheads. To simplify our figures, we omit parked threads from them.

The main thread handling each RPC uses *fork-join* parallelism to fan concurrent requests out to many leaves. The main thread wakes a parked thread to issue each outgoing RPC, blocking on its reply. As replies arrive, these threads decrement a shared atomic counter before parking on a condition variable to track the last reply. The last reply signals the main thread to execute the continuation that merges leaf results and responds to the client.

We next detail each synchronous model with respect to a single RPC execution. For simplicity, our figures show a three-tier service with a single client, mid-tier, and leaf.

Synchronous In-line Block (SIB). This model is the simplest, having only a single thread pool (Fig. 3(a)). *In-line* threads *block* on network sockets awaiting work, and then execute a received RPC to completion, signalling parked threads for outgoing RPCs as needed. The thread pool must grow with higher load.

Synchronous In-line Poll (SIP). SIP differs from SIB in that threads poll for new work using non-blocking APIs (Fig. 3(a)). SIP avoids blocked thread wakeups when work arrives, but, each in-line thread fully utilizes a CPU.

Synchronous Dispatch Block (SDB). SDB comprises two thread pools (Fig. 3(b)). The *network threads* block on socket APIs awaiting new work. But, rather than executing the RPC, they *dispatch* the RPC to a *worker* thread pool by using producer-consumer task-queues and signalling condition variables. Workers pull requests from task queues, and then process them much like the prior in-line threads (i.e., forking for fan-out and issuing syn-

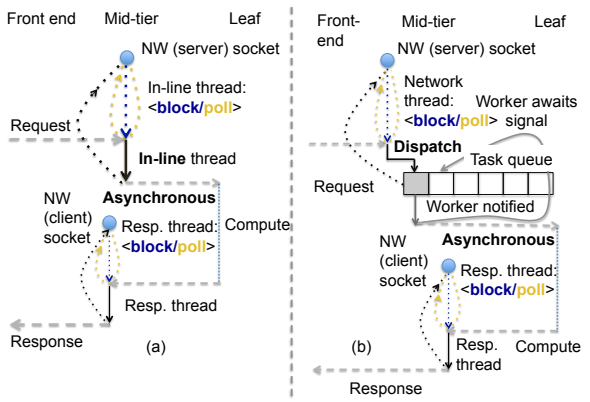


Figure 4: Execution of an RPC by (a) AIB/AIP (b) ADB/ADP

chronous leaf requests). A worker sends the RPC reply to the front-end, before blocking on the condition variable to await new work. Both network and worker pool sizes are variable. Concurrency is limited by the worker pool size. Typically, a single network thread is sufficient.

SDB restricts incoming socket interactions to the network threads, which improves locality; RPC and OS interface data structures do not migrate among threads.

Synchronous Dispatch Poll (SDP). In SDP, network threads *poll* on front-end sockets for new work (Fig. 3(b)).

3.3 Asynchronous models

Asynchronous models differ from synchronous in that they do not tie an execution thread to a specific RPC—all RPC state is explicit. Such models are event-based—an event, such as a leaf request completion, arrives on any thread and is matched to its parent RPC using shared data structures. So, any thread may progress any RPC through its next execution stage. This approach requires drastically fewer thread switches during an RPC lifetime. For example, leaf request fan-outs require a simple for loop, instead of a complex *fork-and-wait*.

To aid non-blocking calls to both leaves and front-end servers, we add another thread pool that exclusively handles leaf server responses—the *response* thread pool.

Asynchronous In-line Block (AIB). AIB (Fig. 4(a)) uses in-line threads to handle incoming front-end requests, and response threads to execute leaf responses. Both thread pools block on their respective sockets awaiting new work. An in-line thread initializes a data structure for an RPC, records the number of leaf responses it expects, records a functor for the continuation to execute when the last response returns, and then fans leaf requests out in a simple for loop. Responses arrive (potentially concurrently) on response threads, which record their results in the RPC data structure and count down until the last response arrives. The final response invokes the continuation to merge responses and complete the RPC.

Asynchronous In-line Poll (AIP). In AIP, in-line and response threads *poll* their respective sockets (Fig. 4(a)).

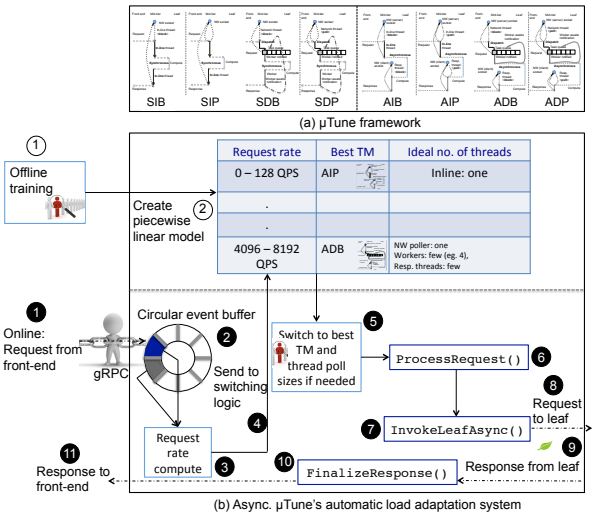


Figure 5: µTune: system design

Asynchronous Dispatch Block (ADB). In ADB, dispatch enables network thread concentration, improving locality and socket contention (Fig. 4(b)). Like SDB, network and worker threads accept and execute RPCs, respectively. Response threads count-down and merge leaf responses. We do not explicitly dispatch responses, as all but the last response thread do negligible work (stashing a response packet and decrementing a counter). All three thread pools vary in size. Typically, one network thread is sufficient, while the other pools must scale with load.

Asynchronous Dispatch Poll (ADP). Network and response threads *poll* for new work (Fig. 4(b)).

4 µTune: System Design

µTune has two features: (a) an implementation of all eight threading models, abstracting RPC (OS/network interactions) within the framework (Fig. 5(a)); and (b) an adaptation system that judiciously tunes threading models under changing load (Fig. 5(b)). *µTune*'s system design challenges include (1) offering a simple interface that abstracts threading from service code, (2) quick load shift detection for efficient dynamic adaptation, (3) adept threading models switches, and (4) sizing thread pools without thread creation, deletion, or management overheads. We discuss how *µTune*'s design meets these challenges.

Framework. *µTune* abstracts the threading model boiler-plate code from service-specific RPC implementation details, wrapping the underlying RPC API. *µTune* enables characterizing the pros and cons of each model.

µTune offers a simple abstraction where service-specific code must implement RPC execution interfaces. For synchronous modes, the service must supply a `ProcessRequest()` method per RPC. `ProcessRequest()` is invoked by in-line or worker threads. This method pre-

pares a concurrent outgoing leaf RPC batch and passes it to `InvokeLeaf()`, which fans it out to leaf nodes. `InvokeLeaf()` returns to `ProcessRequest()` after receiving all leaf replies. The `ProcessRequest()` continuation merges replies and forms a response to the client.

For asynchronous modes, *µTune*'s interface is slightly more complex. Again, the service must supply `ProcessRequest()`, but, it must explicitly represent RPC state in a shared data structure. `ProcessRequest()` may make one/more calls to `InvokeLeafAsync()`. These calls are passed an outgoing RPC batch, a tag identifying the parent RPC, and a `FinalizeResponse()` callback. The tags enable request-response matching. The last arriving response thread invokes `FinalizeReponse()`, which may access the RPC data structure and response protocol buffers from each leaf. A developer must ensure thread-safety. `FinalizeResponse()` may be invoked any time after `InvokeLeafAsync()`, and may be concurrent with `ProcessRequest()`. Reasoning about races is the key challenge of asynchronous RPC implementation.

Automatic load adaptation. A key feature of *µTune* is its ability to automatically select among threading models in response to load, thereby relieving developers of the burden of selecting a threading model a priori.

Synchronous vs. asynchronous microservices have a major programmability gap. Although *µTune*'s framework hides some complexity, it is not possible to switch automatically and dynamically between synchronous and asynchronous modes, as their API and application code requirements necessarily differ. If an asynchronous implementation is available, it will outperform its synchronous counterpart. So, we build *µTune*'s adaption separately for synchronous and asynchronous models.

µTune picks the latency-optimal model among the four options (in-line vs. dispatch; block vs. poll) and tunes thread pool sizes dynamically with load. *µTune* aims to curtail 99^{th}% tail latency. It monitors service load and (a) picks a latency-optimal threading model, then (b) scales thread pools by parking/unparking threads. Both adaptations use profiles generated during an offline training phase. We describe the training and adaptation steps shown in Fig. 5(b).

Training phase. (1) During offline characterization, we use a synthetic load generator to drive specific load levels for sustained intervals. During these intervals, we vary threading model and thread pool sizes and observe 99^{th}% tail latencies. The load generator then ramps load incrementally, and we re-characterize at each load step. (2) *µTune* then builds a piece-wise linear model relating offered load to observed tail latency at each load level.

Runtime adaptation. (1) *µTune* uses event-based windowing to monitor loads offered to the mid-tier at runtime. (2) *µTune* records each request's arrival timestamp in a circular buffer. (3) It then estimates the inter-arrival rate by using the circular buffer's size, and youngest and oldest recorded timestamps. The adaptation system's responsiveness can be tuned by adjusting the circular buffer's size. Careful buffer size tuning can ensure quick, efficient adaptation by avoiding oscillations triggered by outliers. Event-based monitoring can quickly detect precipitous load increases. (4) The inter-arrival rate estimate is then fed as input to the switching logic that interpolates within the piece-wise linear model to estimate tail latency for each configuration under each model and thread pool size. (5) *µTune* then transitions to the predicted lowest latency threading model. *µTune* transitions by "parking" the current threading model and "unparking" the newly selected model using its framework abstraction and condition variable signaling, to (a) alternate between poll/block socket reception, (b) process requests in-line or via predefined task queues that dispatch requests to workers, or (c) park/unpark various thread pools' threads to handle new requests. Successive asynchronous requests invoke the (6) `ProcessRequest()`, (7) `InvokeLeafAsync()`, and (10) `FinalizeResponse()` pipeline as dictated by the new threading model. In-flight requests during transitions are handled by the earlier model.

5 Implementation

Framework. *µTune* builds upon Google's open-source `gRPC` [18] library, which uses *protocol buffers* [33]—a language-independent interface definition language and wire format—to exchange RPCs. *µTune*'s mid-tier framework uses `gRPC`'s C++ APIs: (1) `Next()` and `AsyncNext()` with a zero second timeout are used to respectively block or poll for client requests, (2) `RPCName()` and `AsyncRPCName()` are called via `gRPC`'s `stub` object to send requests to leaves. *µTune*'s asynchronous models explicitly track request state using finite state machines. Asynchronous models' response threads call `Next()` or `AsyncNext()` for block- or poll-based receive.

µTune uses `AsyncRPCName()` to handle asynchronous clients. For asynchronous *µTune*, leaves must use `gRPC`'s `Next()` APIs to accept requests through explicitly managed completion queues; for synchronous, the leaves can use underlying synchronous `gRPC` abstractions.

Using *µTune*'s framework to build a new microservice is simple, as only a few service specific functions must be defined. We took ∼2 days for each service in Sec. 6.

Automatic load adaptation. We construct the piecewise linear model of tail latency by averaging five 30s measurements of each threading model-thread pool pair at varying loads. *µTune*'s load detection relies on a thread-safe circular buffer built using scoped locks and condition variables. The circular buffer capacity is tuned to quickly detect load transients while avoiding oscillation. We use a 5-entry circular buffer in all experiments. *µTune*'s switching logic uses C++ atomics and condition variables to

switch among threading models seamlessly. *µTune*'s adaptation code spans 2371 LOC of C++.

6 Experimental Setup

We characterize threading models in the context of four information retrieval OLDI applications' mid-tier and leaf microservices adopted from µSuite [134].

HDSearch. `HDSearch` performs content-based image similarity search by matching nearest neighbors (NN) in a high-dimensional feature space. It serves a 500K image corpus from Google's Open Images data set [30]. Each image is indexed via a 2048-dimensional feature vector created using Google's Inception V3 model [136] implemented in `TensorFlow` [42]. `HDSearch` locates response images whose feature vectors are near the query's [65,96].

Mid-tier microservice. Modern k-NN libraries use indexing structures, such as Locality-Sensitive Hash (LSH) tables, kd-trees, or k-means, to reduce exponentially the search space relative to brute-force linear search [44, 52, 75,85,110,129,137–139]. `HDSearch`'s mid-tier uses LSH (an accurate and fast algorithm [45, 62, 67, 131]) via an open-source k-NN library called Fast Library for Approximate Nearest Neighbors (FLANN) [115]. The mid-tier's LSH tables store {leaf-server, point id} tuples indicating feature vectors in the leaf's data shards. While executing RPCs, the mid-tier probes its in-memory LSH tables to gather potential NNs. It then sends RPCs with potential NN point IDs to the leaves. Leaves compute distances to return a distance-sorted list. The mid-tier merges leaf responses to return the k-NN across all shards.

Leaf microservice. The leaf's distance computations are embarrassingly parallel, and can be accelerated with SIMD, multi-threading, and distributed computing [65]. We employ all techniques. We distribute distance computations over multiple leaves until the distance computation time and network communication overheads are roughly balanced. Hence, the mid-tier's latency, and its ability to fan out RPCs quickly, becomes critical: the mid-tier microservice and network overheads limit the leaf microservice's scalability. Leaves compare query feature vectors against point lists received from the mid-tier using the high-accuracy Euclidean distance metric [75].

Router. `Router` performs replication-based protocol routing for scaling fault-tolerant key-value stores. Queries are `get` or `set` requests. `Gets` contain keys, and return the corresponding value. `Sets` contain key-value pairs, and return a `set` completion acknowledgement. `Get` and `set` query distributions mimic YCSB's Workload A [59] (1:1 ratio). Queries are from a "Twitter" data set [71].

Mid-tier microservice. The mid-tier uses SpookyHash [8] to distribute keys uniformly across leaf microservers and route `get` and `set` queries. `Router` replicates data for better availability, allowing the same data to reside on several leaves. The mid-tier routes `sets` to all replicas and distributes `gets` among replicas. The mid-tier merges leaf responses and sends them to the client.

Leaf microservice. The leaf microserver builds a `gRPC`-based communication wrapper around a memcached [72] instance, exporting `get` and `set` RPCs.

Set Algebra. `Set Algebra` performs document search by intersecting posting lists. It searches a corpus of 4.3 million WikiText documents in Wikipedia [40] sharded uniformly across leaf microservers, to identify documents containing all search terms. Leaf microservers index posting lists for each term in their shard of the document corpus. Stop words determined by collection frequency [149] are excluded from the term index to reduce leaf computation. Search queries (typically a series of ≤ 10 words [4]) are synthetically generated based on the probability of word occurrences in Wikipedia [40].

Mid-tier microservice. The mid-tier forwards client queries containing search terms to the leaf microservers, which then return intersected posting lists to the mid-tier for their respective shards. The mid-tier aggregates the per-shard posting lists and returns their union to the client.

Leaf microservice. Leaves look up posting lists for all search terms and then intersect the sorted lists. The resulting intersection is returned to the mid-tier.

Recommend. `Recommend` is a recommendation service that performs user-based collaborative filtering on a data set of 10K {user, item, rating} tuples—derived from the `MovieLens` movie recommendation data set [78]—to predict a user's rating for an item. The data set is sharded equally among leaves. `Recommend` uses a fast, flexible open-source ML library called `mlpack` [60] to perform collaborative filtering using matrix decomposition.

Mid-tier microservice. The mid-tier gets {user, item} query pairs and forwards them to the leaves. Item ratings sent by the leaves are averaged and sent to the client.

Leaf microservice. Leaves perform collaborative filtering on a pre-composed matrix of {user,item,rating} tuples. Rating predictions are then sent to the mid-tier.

We use a load generator that mimics many clients to send queries to each mid-tier microservice under controlled load scenarios. It operates in a closed-loop mode while measuring peak sustainable throughput. We measure end-to-end (across all microservices) 99^{th}% latency by operating the load generator in open-loop mode with Poisson inter-arrivals [57]. The load generator runs on separate hardware and we validated that the load generator and network bandwidth are not performance bottlenecks.

Our distributed system has a load generator, a mid-tier microservice, and (1) four-way sharded leaf microservice for `HDSearch`, `Set Algebra`, and `Recommend` and (2) 16-way sharded leaf microservice with three replicas for `Router`. The hardware configuration of our measurement setup is in Table 1. The leaf microservers run within

Table 1: Mid-tier microservice hardware specification.

Processor	Intel Xeon E5-2699 v3 "Haswell"
Clock frequency	2.30 GHz
Cores / HW threads	36 / 72
DRAM	500 GB
Network	10Gbit/s
Linux kernel version	3.19.0

Linux `tasksets` limiting them to 20 logical cores for `HDSearch`, `Set Algebra`, and `Recommend` and 5 logical cores for `Router`. Each microservice runs on a dedicated machine. The mid-tier is not CPU bound; saturation throughput is limited by leaf server CPU.

To test the effectiveness of *µTune*'s load adaptation system and measure its responsiveness to load changes, we construct the following load generator scenarios. (1) *Load ramp*: We increase offered load in discrete 30s steps from 20 Queries Per Second (QPS) up to a microservice-specific near-saturation load. (2) *Flash crowd*: We increase load suddenly from 100 QPS to 8K/13K QPS. In addition to performance metrics measured by our load generator, we also report OS and microarchitectural statistics. We use Linux's `perf` utility to profile the number of cache misses and context switches incurred by the mid-tier microservice. We use Intel's HITM (hit-Modified) PEBS coherence event to detect true sharing of cache lines; an increase in HITM events indicates a corresponding increase in lock contention [109]. We measure thread wakeup delays (reported as latency histograms) using the BPF run queue (scheduler) latency tool [23].

7 Evaluation

We first characterize our threading models. We then compare *µTune* to state-of-the-art adaptation systems.

7.1 Threading model characterization

We explore microservice threading models by first comparing synchronous vs. asynchronous performance. We then separately explore trade-offs among the synchronous and asynchronous models to report how the latency-optimal threading model varies with load.

7.1.1 Synchronous vs. Asynchronous

The synchronous vs. asynchronous trade-off is one of programmability vs. performance. It would be unusual for a development team to construct both microservice designs; if the team invests in the asynchronous design, it will almost certainly be more performant. Still, our performance study serves to quantify this gap.

Saturation throughput. We record saturation throughput for the "best" threading model at saturation (SDB/ADB). In Fig. 6, we see that the greater asynchronous efficiency improves saturation throughput for *µTune*'s asynchronous models, a 42% mean throughput

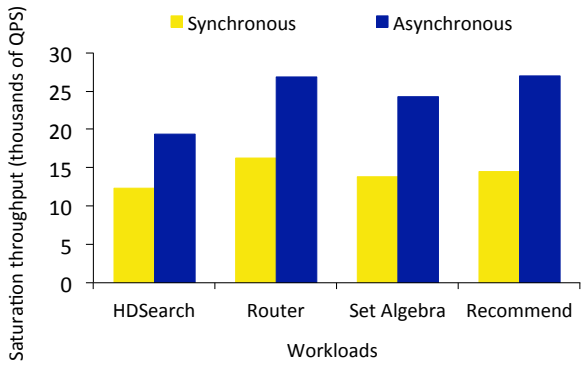


Figure 6: Sync. vs. async. saturation throughput: async. does better by a mean 42%.

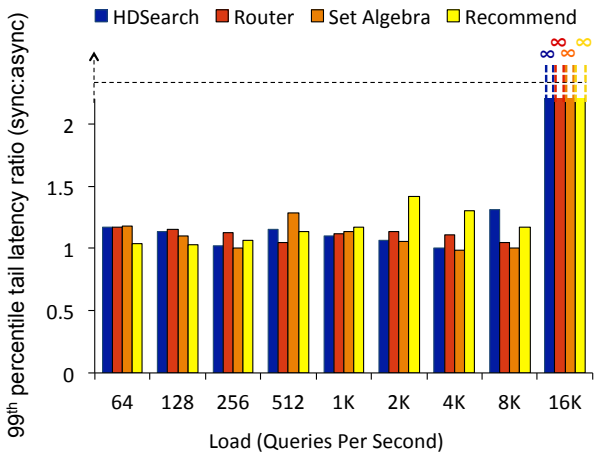


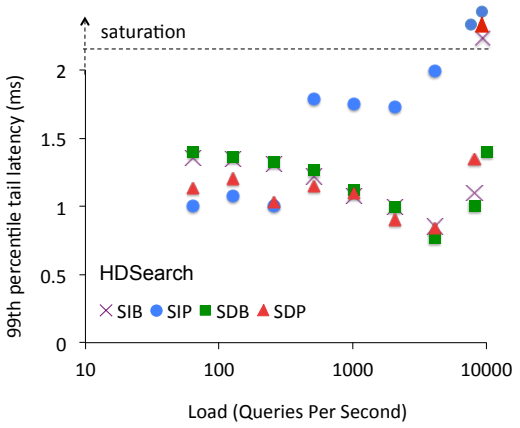
Figure 7: Best sync:async tail latency ratio: async. is faster by a mean 12% at sync.-achievable loads & infinitely faster at high loads.

boost across all services. But, we spent 5× more effort to build, debug, and tune the asynchronous models.

Tail latency. Latency cannot meaningfully be measured at saturation, as the offered load is unsustainable and queuing delays grow unbounded. So, we compare tail latencies at load levels from 64 QPS up to synchronous saturation. In Fig. 7, we show the best sync-to-async ratio of 99^{th}% tail latency across all threading models and thread pool sizes at each load level; we study inter-model latencies later. We find asynchronous models improve tail latency up to $\sim 1.3\times$ (mean of $\sim 1.12\times$) over synchronous models (for loads that synchronous models can sustain; i.e., $\leq$ 8K). This substantial tail latency gap arises because asynchronous models prevent long queuing delays.

7.1.2 Synchronous models

We study the tail latency vs. load trade-off for services built with *µTune*'s synchronous models. We show a cross-product of the threading taxonomy across loads for `HDSearch` in Fig. 8. Each data point is the best 99^{th}% tail latency for that threading model and load based on an exhaustive thread pool size search. Points above the dashed



QPS	64	128	256	512	1024	2048	4096	8192	10K
SIB	1.4	1.3	1.3	**1**	**1**	**1**	1.1	1.1	∞
SIP	**1**	**1**	**1**	1.6	1.6	1.9	2.6	∞	∞
SDB	1.4	1.3	1.3	1.1	1.1	1.1	**1**	**1**	**1**
SDP	1.2	1.1	**1**	**1**	**1**	**1**	1.1	1.4	∞

Figure 8: Graph: Latency vs. load trade-off for `HDSearch` sync. models. Table: Latencies at each load normalized to the best latency for that load—No threading model is always the best.

line are in saturation, where tail latencies are very high and meaningless. The table reports the same graph data with each load latency normalized to the best latency for that load, which is highlighted in blue. We omit graphs for other applications as they match the `HDSearch` trends.

We make the following observations:

SDB enables highest load. SDB, with a single network thread and a large worker pool of 50 threads is the only model that sustains peak loads ($\geq$ 10K QPS). SDB is best at high loads as (1) its worker pool has enough concurrency so that leaf microservers, rather than the mid-tier, pose the bottleneck; and (2) the single network thread is enough to accept and dispatch the offered load. SDB outperforms SDP at high load as polling consumes CPU in fruitless poll loops. For example, at 10,000 QPS, the mid-tier microserver receives one query every 100 microseconds. In SDP, poll loops are often shorter than 100 microseconds. Hence, some poll loops that do not retrieve any requests are wasted work and may delay critical work scheduling, such as RPC response processing. Under SDB, the CPU time wasted in empty poll loops can instead be used to progress an ongoing request.

SIP has lowest latency at low load. While SDB sustains peak loads, it is latency-suboptimal at low loads. SIP offers 1.4× better low-load tail latency by avoiding up to two OS thread wakeups relative to alternative models: (1) network thread wakeups via interrupts on query arrivals, and (2) worker wakeups for RPC dispatch. Work hand-off among threads may cause OS-induced scheduling tails.

SDP is best at intermediate loads. SIP ceases being the best model when the offered load grows too large for

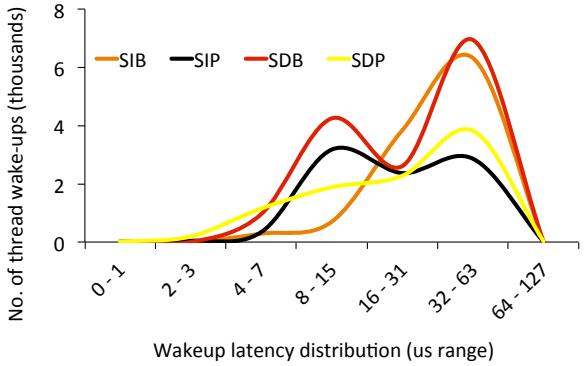


Figure 9: `HDSearch` sync. thread wakeups at 64 QPS: Block incurs more wakeups.

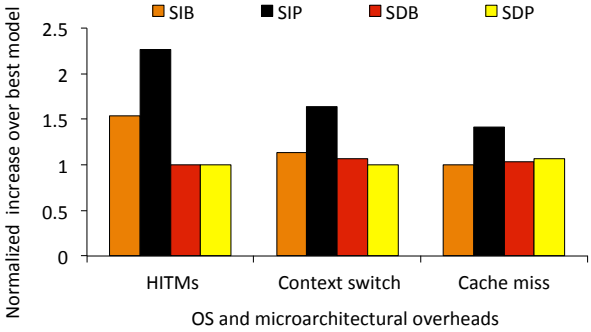
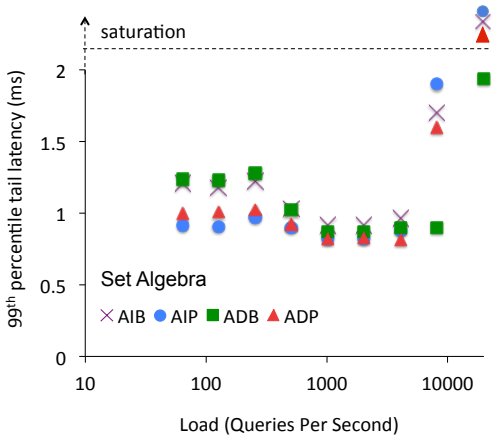


Figure 10: Relative frequency of sync. contention, context switches & cache misses at 10K QPS: SIP does worst.

one in-line thread to sustain. Adding more in-line polling threads causes contention in the OS and RPC reception code paths. Additional in-line blocking threads are less disruptive, but SIB never outperforms SDP. By switching to a dispatched model, a single network thread can still accept the incoming RPCs, avoiding contention and locality losses of running the `gRPC` [18] and network receive stacks across many cores. The workers add sufficient concurrency to sustain RPC and response processing. We further note that SDP tail latencies at intermediate loads are better than at low load, since there is better temporal locality and OS and networking performance tend to improve due to batching effects in the networking stack.

OS and microarchitectural effects. We report OS thread wakeup latency distributions for `HDSearch` synchronous models at 64 QPS in Fig. 9. Although some OS thread wakeups are fast (∼5 μs), blocking models frequently incur 32-64 μs range wakeups. This data also depicts the advantage of in-line over dispatched models with respect to low-load worker wakeup costs.

Fig. 10 shows the relative frequency of true sharing misses (HITM), context switches, and cache misses for threading models at high load (10K QPS). These results show why SIP fails to scale as load increases. SIP needs multiple threads to sustain loads $\geq$ 512 QPS. Multiple pollers contend pathologically on the network receive pro-



QPS	64	128	256	512	1024	2048	4096	8192	20K
AIB	1.3	1.3	1.3	1.2	1.1	1.1	1.2	1.9	∞
AIP	**1**	**1**	**1**	**1**	**1**	**1**	1.1	2.1	∞
ADB	1.4	1.4	1.3	1.1	1.1	1.1	1.1	**1**	**1**
ADP	1.1	1.1	1.1	**1**	**1**	**1**	**1**	1.8	∞

Figure 11: Graph: Latency vs. load for `Set Algebra` async. models. Table: Latencies at each load normalized to the best latency for that load—No threading model is always the best.

cessing, incurring many sharing misses, context switches, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar contention. However, SDB outperforms SDP, since SDP incurs a mean $\sim$ 10% higher wasted CPU utilization.

Additional Tests. (1) We measured *µTune* with null (empty) RPC handlers. Complete services incur higher tails than null RPCs as mid-tier and leaf computations add to tails. For null RPCs, SIP outperforms SDB by 1.57× at low loads. (2) We measured `HDSearch` on another hardware platform (Intel Xeon "Skylake" vs. "Haswell"). We notice similar trends as on our primary Haswell platform, with SIP outperforming SDB by 1.42× at low loads. (3) We note that the median latency follows a similar trend, but, with lower absolute values (e.g., `HDSearch`'s SIP outperforms SDB by 1.26× at low load). We omit figures for these tests as they match the reported `HDSearch` trends. Threading performance gaps will be wider for faster services (e.g., 200K QPS Memcached [26]) as slightest OS/network overheads will become magnified [122].

7.1.3 *Asynchronous models*

We show results for `Set Algebra`'s asynchronous models in Fig. 11. As above, we omit figures for additional services as they match `Set Algebra` trends. Broadly, trends follow the synchronous models, but latencies are markedly lower. We note the following differences:

Smaller thread pool sizes. Significantly smaller (≤ 4 threads) thread pool sizes are sufficient at various loads, since asynchronous models capitalize on the available concurrency by quickly moving on to successive requests.

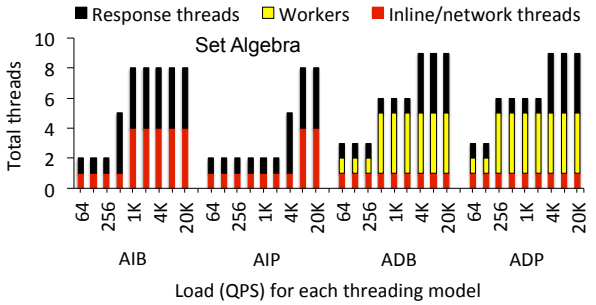


Figure 12: Async. thread pools for best tails: Big pools contend.

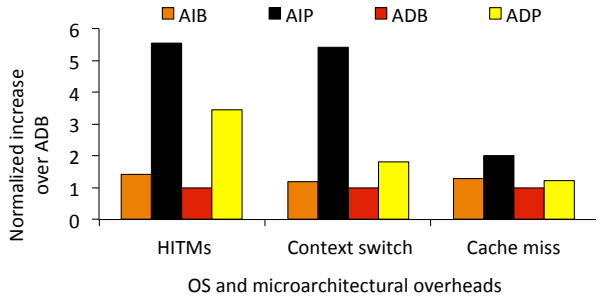


Figure 13: Async. `Set Algebra`'s relative frequency of contention, context switches, & cache misses over best model at peak load: AIP performs worst.

Fig. 12 shows `Set Algebra`'s asynchronous thread pool sizes that achieve the best tails for each load level. We find four threads enough to sustain high loads. Larger thread pools deteriorate latency by contending for network sockets or CPU resources. In contrast, SIB, SDB, and SDP need many threads (as many as 50) to exploit available concurrency.

AIP scales much better than SIP. AIP with just one in-line and response thread can tolerate much higher load (up to 4096 QPS) than SIP, since queuing delays engendered by both the front-end network socket and leaf node response sockets are avoided by the asynchronous design.

ADP scales worse than SDP. ADP with 4 worker and response threads copes worse than SDP at loads $\geq$ 8192 QPS even though it does not have a large thread pool contending for CPU (in contrast to SDP at high loads). This design fails to scale since response threads contend on the completion queue tied to leaf node response sockets.

OS and microarchitectural effects. Unlike SDP, ADP incurs more context switches, caches misses, and HITMs, due to response thread contention (Fig. 13).

7.2 Load adaptation

We next compare *µTune*'s load adaptation against state-of-the-art baselines [43, 76, 97] for various load patterns.

7.2.1 *Comparison to the state-of-the-art*

We compare *µTune*'s run-time performance to state-of-the-art adaptation techniques [43, 76, 97]. We find that

μ*Tune* offers better tail latency than these approaches.

Few-to-Many (FM) parallelism. FM [76] uses offline profiling to vary parallelism during a query's execution. The FM scheduler decides *when* to add parallelism for long-running queries and by *how much*, based on the dynamic load that is observed every 5 ms. In consultation with FM's authors, we opt to treat a microservice as an FM query, to create a fair performance analogy between μ*Tune* and FM. In our FM setup, we mimic FM's offline profiling by building an offline interval table that notes the software parallelism to add for varied loads in terms of thread pool sizes. We use the peak load-sustaining synchronous and asynchronous models (SDB and ADB). During run-time, we track the mid-tier's loads every 5 ms and suitably vary SDB/ADB's thread pool sizes. FM varies only pool sizes (vs. μ*Tune* also varying threading models), and we find that FM underperforms μ*Tune*.

Integrating Polling and Interrupts (IPI). Langendoen et al. [97] propose a user-level communication system that adapts between poll- and interrupt-driven request reception. The system initially uses interrupts. It starts to poll when all threads are blocked. It reverts to interrupts when a blocked thread becomes active. We study this system for synchronous modes only; as its authors note [97], it does not readily apply for asynchronous modes.

To implement this technique, we keep (1) a global count of all threads, and (2) a shared atomic count of *blocked* threads for the mid-tier. Before a thread becomes *blocked* (e.g., invokes a synchronous call), it increments the shared count and decrements it when it becomes active (i.e., synchronous call returns). After revising the shared count, a thread checks if the system's *active* thread count exceeds the machine's logical core count. If higher, the system blocks, otherwise, it shifts to polling. We find that μ*Tune* outperforms this technique, as it considers additional model dimensions (such as inline/dispatch), as well as dynamically scales thread pools based on load.

Time window-Based Detection (TBD). Abdelzaher *et al.* [43] periodically observe request arrival times in fixed observation windows to track request rate. In our setup, we replace μ*Tune*'s event-based detector with this time-based detector. We pick 5 ms time-windows (like FM) to track low loads and react quickly to load spikes.

We evaluate the tail latency exhibited by μ*Tune* across all services, and compare it to these state-of-the-art approaches [43, 76, 97] for both steady-state and transient loads. We examine μ*Tune*'s ability to pick a suitable threading model and size thread pools for time-varying load. We offer loads that differ from those used in training. We aim to study if μ*Tune* selects the best threading model, as compared to an offline exhaustive search.

7.2.2 *Steady-state adaptation*

Fig. 14 shows μ*Tune*'s ability in converging to the best threading model and thread pool size for steady-state loads. Our test steps up and down through the displayed load levels. We report the tail latency at each load averaged over five trials. The SIP1, SDP1-20, and SDB1-50 bars are optimal threading configurations for some loads. The nomenclature is the threading model followed by the pool sizes, in the form model-network-worker-response. The FM [76], Integrated Poll/Interrupt (IPI) [97], and Time-Based Detection (TBD) [43] bars are the tail latency of state-of-the-art systems. The red bars are μ*Tune*'s tail latency; bars are labelled with the configuration μ*Tune* chose.

In synchronous mode (Fig. 14 (top)), μ*Tune* first selects an SIP model with a single thread, until load grows to about 1K QPS, at which point it switches to SDP, and begins ramping up the worker thread pool size. At 8K QPS, it switches to SDB and continues growing the worker thread pool, until it reaches 50 threads, which is sufficient to meet the peak load the leaf microservice can sustain.

μ*Tune* boosts tail latency by up to 1.7× for `HDSearch`, 1.6× for `Router`, 1.4× for `Set Algebra`, and 1.5× for `Recommend` (at 20 QPS) over SDB—the static model that sustains peak loads. μ*Tune* boosts tail latency by a mean 1.3x over SDB across all loads and services. μ*Tune* also outperforms all state-of-the-art [43, 76, 97] techniques (except TBD) for at least one load level and never underperforms. μ*Tune* outperforms FM by up to 1.3× for `HDSearch` and `Recommend`, and 1.4× for `Router` and `Set Algebra` under low loads, as FM only varies SDB's thread pool sizes and hence incurs high network poller and worker wakeups. μ*Tune* outperforms the IPI approach by up to 1.6× for `HDSearch`, 1.5× for `Router` and `Recommend`, and 1.4× for `Set Algebra` under low loads. At low load, IPI polls with many threads (to sustain peak load), succumbing to expensive contention. TBD does as well as μ*Tune* as the requests mishandled during the 5 ms monitor window fall in tails greater than the 99^{th}% percentile that we monitor for 30s for each load level.

In asynchronous mode (Fig. 14 (bottom)), μ*Tune* again initially selects an in-line poll model with small-sized pools, transitioning to ADP and then ADB as load grows. Four worker and response threads suffice for all loads. We show that μ*Tune* outperforms static threading choices and state-of-the-art techniques by up to 1.9× for at least one load level.

Across all loads, μ*Tune* selects threading models and thread pool sizes that perform within 5% of the best model as determined by offline search. μ*Tune* incurs less than 5% mean instruction overhead over the load-specific "best" threading model, as depicted in Fig. 15. Hence, we find our piece-wise linear model sufficient to make good threading decisions. Note that μ*Tune* always prefers a

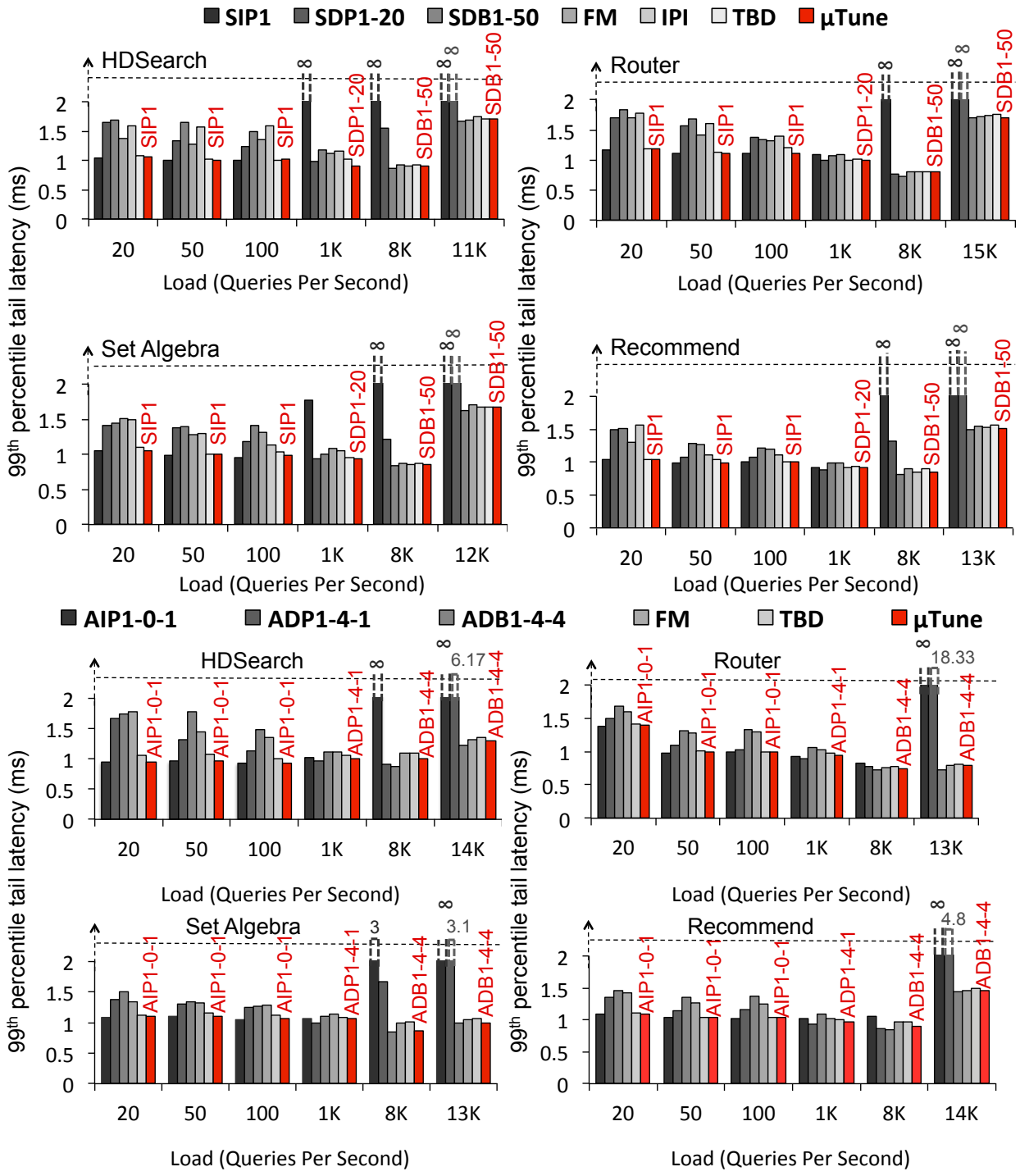


Figure 14: Synchrounous (top) & asynchronous (bottom) steady-state adaptation.

single thread interacting with the front-end socket. This finding underscores the importance of maximizing locality and avoiding contention on the RPC receive path.

7.2.3 *Load transients*

Table 2 indicates *µTune*'s response to load transients, where the columns are a series of varied-duration load levels. The rows are the 99^{th}% tail latency for the models between which *µTune* adapts in this scenario (SIP/AIP

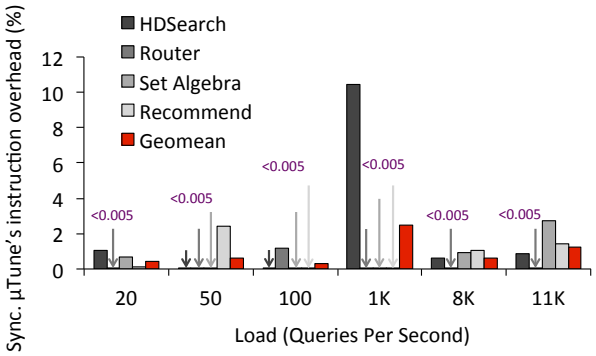


Figure 15: Sync. *µTune*'s instruction overhead for steady-state loads: less than 5% mean overhead incurred.

		Synchronous				Asynchronous		
		100 QPS (0 - 30s)	8K QPS (30s - 31s)	100 QPS (31 - 61s)		100 QPS (0 - 30s)	13K QPS (30s - 31s)	100 QPS (31 - 61s)
HDSearch	SIP	0.99	>1s	>1s	AIP	0.95	>1s	>1s
	SDB	1.49	1.07	1.40	ADB	1.48	1.10	1.40
	FM	1.35	13.00	1.32	FM	1.28	4.73	1.33
	IPI	1.59	1.10	1.50	IPI	NA	NA	NA
	TBD	1.03	8.69	1.02	TBD	1.06	2.63	1.08
	µTune	1.01	1.09	0.99	*µTune*	0.98	1.13	0.96
Router	SIP	1.10	>1s	>1s	AIP	1.01	>1s	>1s
	SDB	1.31	0.83	1.36	ADB	1.35	1.13	1.31
	FM	1.33	9.40	1.40	FM	1.30	12.95	1.30
	IPI	1.4	1.10	1.38	IPI	NA	NA	NA
	TBD	1.13	4.51	1.11	TBD	1.03	6.24	1.01
	µTune	1.12	0.88	1.13	*µTune*	0.99	1.02	0.98
Set Algebra	SIP	0.95	>1s	>1s	AIP	1.04	>1s	>1s
	SDB	1.30	0.92	1.32	ADB	1.26	0.99	1.23
	FM	1.30	12.00	1.25	FM	1.28	4.14	1.27
	IPI	1.20	0.94	1.12	IPI	NA	NA	NA
	TBD	1.00	8.45	1.03	TBD	1.09	6.62	1.1
	µTune	0.97	0.92	1.03	*µTune*	1.06	1.1	1.06
Recommend	SIP	1.00	>1s	>1s	AIP	1.03	>1s	>1s
	SDB	1.26	0.96	1.22	ADB	1.37	1.30	1.32
	FM	1.23	>1s	>1s	FM	1.28	8.61	1.20
	IPI	1.13	1.02	1.13	IPI	NA	NA	NA
	TBD	1.02	4.96	1.03	TBD	1.06	6.00	1.07
	µTune	1.00	1.00	1.00	*µTune*	1.06	1.39	1.04

Table 2: 99^{th}% tail latency (ms) for load transients.

and SDB/ADB), state-of-the-art [43, 76, 97] techniques, and *µTune*. The key step in this scenario is the 8K/13K QPS load level, which lasts only 1s. We pick spikes of 8K QPS and 13K QPS for synchronous and asynchronous as these loads are SIP and AIP saturation levels, respectively.

We find that the in-line poll models accumulate a large backlog during the transient as they saturate, and thus perform poorly even during successive low loads. FM and TBD incur high transient tail latencies as they allow requests during the 5 ms load detection window to be handled by sub-optimal threading choices. FM saturates at 8K QPS for `Recommend` since the small SDB thread pool size opted by FM at 100 QPS causes unbounded queuing during the load monitoring window. IPI works only for synchronous and performs poorly at low loads as its fixed-size thread pool leads to polling contention. We show that *µTune* detects the transient and transitions from SIP/AIP to SDB/ADB fast enough to avoid accumulating a backlog that affects tail latency. Once the flash crowd subsides, *µTune* transitions back to SIP/AIP, avoiding the latency penalty SDB/ADB suffer at low load.

8 Discussion

We briefly discuss open questions and *µTune* limitations.

Offline training. *µTune* uses offline training to build a piece-wise linear model. This phase might be removed by analyzing dynamically OS and hardware signals, such as context switches, thread wakeups, queue depths, cache misses, and lock contention, to switch threading models. Designing heuristics to switch optimally based on such run-time metrics remains an open question; our performance characterization can help guide their development.

Thread pool sizing. *µTune* tunes thread pool sizes using a piece-wise linear model. *µTune* differs from prior thread pool adaptation systems [76, 86, 93] in that it also tunes threading models. Some of these systems use more sophisticated tuning heuristics, but we did not observe opportunity for further improvement in our microservices.

CPU cost of polling. *µTune* polls at low loads to avoid thread wakeups. Polling can be costly as it wastes CPU time in fruitless poll loops. However, as most operators over-provision CPU to sustain high loads [130], when load is low, spare CPU time is typically available [79].

***µTune*'s asynchronous framework.** Asynchronous RPC state must be maintained in thread-safe structures, which is challenging. More library/language support might simplify building asynchronous microservices with *µTune*. We leave such support to future work.

Comparison with optimized systems that use kernel-bypass, multi-queue NICs, etc. It may be interesting to study the implications of optimized systems [53, 87, 91, 103, 121, 122] that incorporate kernel-bypass, multi-queue NICs, etc., on threading models and *µTune*. Multi-queue NICs may improve polling scalability; multiple network pollers currently contend for the underlying `gRPC` [18] queues under *µTune*. OS-bypass may further increase the application threading model's importance; for example, it may magnify the trade-off between in-line and dispatch RPC execution, as OS-bypass eliminates latency and thread hops in the OS TCP/IP stack, shifting the break-even point to favor in-line execution for longer RPCs. However, in this paper, we have limited our scope to study designs that can layer upon (unmodified) `gRPC` [18]; we defer studies that require extensive `gRPC` [18] changes (or an alternative reliable transport) to future work.

9 Related Work

We discuss several categories of related work.

Web server architectures. Web servers can have (a) thread-per-connection [119], (b) event-driven [120], (c) thread-per-request [84], or (d) thread-pool architectures [104]. Pai *et al.* [119] build thread-per-connection servers as multi-threaded processes. Knot [145] is a thread-per-connection non-blocking server. In contrast, μ*Tune* is a thread-per-request thread-pool architecture that scales better for microservices [104]. The Single Process Event-Driven (SPED) [119] architecture operates on asynchronous ready sockets. In contrast, μ*Tune* supports both synchronous and asynchronous I/O. The SYmmetric Multi-Process Event-Driven (SYMPED) [120] architecture runs many processes as SPED servers via context switches. The Staged Event-Driven Architecture (SEDA) [148] joins event-driven stages via queues. A stage's thread pool is driven by a resource controller. Apart from considering synchronous and asynchronous I/O like prior works [69,83,84,120,146,148], μ*Tune* also studies a full microservice threading model taxonomy. gRPC-based systems such as Envoy [13] or Finagle [16] act as load balancers or use a single threading model.

Software techniques for tail latency: Prior works [84, 148] note that monolithic service software designs can significantly impact performance. But, microsecond–scale OS and network overheads that dominate in μ*Tune*'s regime do not manifest in these slower services. Some works improve web server software via software pattern re-use [127, 128], caching file systems [84], or varying parallelism [76], all of which are orthogonal to the questions we investigate. Kapoor *et al.* [91] also note that OS and network overheads impact short-running cloud services, but, their kernel bypass solution may not apply for all contexts (e.g., a shared cloud infrastructure).

Parallelization to reduce latency: Several prior works [54,61,89,94,99,104,124,135,147] reduce tails via parallelization. Others [47,70,80,112] improve medians by adaptively sharing resources. Prior works use prediction [86,93], hardware parallelism [76], or data parallelism [73] to reduce monolithic services' tail latency. Lee *et al.* [99] use offline analysis (like μ*Tune*) to tune thread pools. We study microservice threading, and vary threading models altogether. But, we build on prior works' thread pool sizing insights.

Hardware mechanisms for tail latency: Several other prior works improve leaf service tail latency via better co-location [102], voltage boosting [82,92], or applying heterogeneity in multi-cores [77]. But, they do not study microservice tail latency effects engendered by software threading, OS, or network.

10 Conclusion

Prior works study monolithic OLDI services, where microsecond-scale overheads are negligible. The rapid advent of faster I/O and low-latency microservices calls for analyzing threading effects for the microsecond regime. In this paper, we presented a structured threading model taxonomy for microservices. We identified different models' performance effects under diverse load. We proposed μ*Tune*—a novel framework that abstracts microservice threading from application code and automatically adapts to offered load. We show that selecting load-optimal threading models can improve tail latency by up to 1.9x.

11 Acknowledgement

We thank our shepherd, Dr. Hakim Weatherspoon, and the anonymous reviewers for their valuable feedback. We thank Amlan Nayak for creating the HDSearch data set, and Neha Agarwal for reviewing μ*Tune*'s code base.

We acknowledge Ricardo Bianchini, Manos Kapritsos, Mosharaf Chowdhury, Baris Kasikci, David Devecsery, Md Haque, Aasheesh Kolli, Karin Strauss, Inigo Goiri, Geoffrey Blake, Joel Emer, Gabriel Loh, A. V. Madhavapeddy, Zahra Tarkhani, and Eric Chung for their insightful suggestions that helped improve this work.

We are especially grateful to Vaibhav Gogte and Animesh Jain for their input in shaping the μ*Tune* concept. We thank P.R. Sriraman, Rajee Sriraman, Amrit Gopal, Akshay Sriraman, Vidushi Goyal, and Amritha Varshini for proof-reading our draft.

This work was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This work was also supported by NSF Grant IIS1539011 and gifts from Intel.

12 References

[1] Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Aerospike. https://www.aerospike.com/docs/client/java/usage/async/index.html.

[3] Apache http server project. https://httpd.apache.org/.

[4] Average number of search terms for online search queries in the United States as of August 2017. https://www.statista.com/statistics/269740/number-of-search-terms-in-internet-research-in-the-us/.

[5] Azure Synchronous I/O antipattern. https://docs.microsoft.com/en-us/azure/architecture/resiliency/high-availability-azure-applications.

[6] The biggest thing amazon got right: The platform. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

[7] BLPOP key timeout. https://redis.io/commands/blpop.

[8] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html.

[9] Building products at soundcloud: Dealing with the monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[10] Building Scalable and Resilient Web Applications on Google Cloud Platform. https://cloud.google.com/solutions/scalable-and-resilient-apps.

[11] Celery: Distributed Task Queue. http://www.celeryproject.org/.

[12] Chasing the bottleneck: True story about fighting thread contention in your code. https://blogs.mulesoft.com/biz/news/chasing-the-bottleneck-true-story-about-fighting-thread-contention-in-your-code/.

[13] Envoy. https://www.envoyproxy.io/.

[14] Facebook Thrift. https://github.com/facebook/fbthrift.

[15] Fighting spam with haskell. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/.

[16] Finagle. https://twitter.github.io/finagle/guide/index.html.

[17] From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. https://www.infoq.com/presentations/linkedin-microservices-urn.

[18] gRPC. https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md.

[19] Handling 1 Million Requests per Minute with Go. http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/.

[20] Improve Application Performance With SwingWorker in Java SE 6. http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html.

[21] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/blog/2009/7/25/latency-iseverywhere-and-it-costs-you-sales-how-to-crush-it.html.

[22] Let's look at Dispatch Timeout Handling in WebSphere Application Server for z/OS. https://www.ibm.com/developerworks/community/blogs/aimsupport/entry/dispatch_timeout_handling_in_websphere_application_server_for_zos?lang=en.

[23] Linux bcc/BPF Run Queue (Scheduler) Latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[24] LPOP key. https://redis.io/commands/lpop.

[25] Mcrouter. https://github.com/facebook/mcrouter.

[26] Memcached performance. https://github.com/memcached/memcached/wiki/Performance.

[27] Microsoft Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[28] mongoDB. https://www.mongodb.com/.

[29] Myrocks: A space- and write-optimized MySQL database. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/.

[30] OpenImages: A public dataset for large-scale multi-label and multi-class image classification. https://github.com/openimages.

[31] Pokemon go now the biggest mobile game in US history. http://www.cnbc.com/2016/07/13/pokemon-go-now-the-biggest-mobile-game-in-us-history.html.

[32] Programmer's Guide, Release 2.0.0. https://www.intel.com/content/dam/www/public/us/en/documents/guides/dpdk-programmers-guide.pdf.

[33] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[34] Redis Replication. https://redis.io/topics/replication.

[35] Resque. https://github.com/defunkt/resque.

[36] RQ. http://python-rq.org/.

[37] Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. https://www.infoq.com/presentations/scale-gilt.

[38] Setting Up Internal Load Balancing. https://cloud.google.com/compute/docs/load-balancing/internal/.

[39] What is microservices architecture? https://smartbear.com/learn/api-design/what-are-microservices/.

[40] Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350.

[41] Workers inside unit tests. http://python-rq.org/docs/testing/.

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Computing Research Repository*, 2016.

[43] T. F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, 1999.

[44] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science*, 2006.

[45] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems*. 2015.

[46] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[47] N. Bansal, K. Dhamdhere, J. Könemann, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 2004.

[48] M. Barhamgi, D. Benslimane, and B. Medjahed. A query rewriting approach for web service composition. *IEEE Transactions on Services Computing*, 2010.

[49] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.

[50] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. In *IEEE Micro*, 2003.

[51] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 2007.

[52] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *International conference on World Wide Web*, 2005.

[53] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Conference on Operating Systems Design and Implementation*, 2014.

[54] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 2007.

[55] A. Bouch, N. Bhatti, and A. Kuchinsky. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *ACM Conference on Human Factors and Computing Systems*, 2000.

[56] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, 2013.

[57] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *International Conference on Telecommunications*. IEEE.

[58] J. L. Carlson. *Redis in Action*. 2013.

[59] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*, 2010.

[60] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 2013.

[61] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Annual International conference on Supercomputing*, 2006.

[62] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Annual Symposium on Computational Geometry*, 2004.

[63] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[64] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[65] C. C. Del Mundo, V. T. Lee, L. Ceze, and M. Oskin. NCAM: Near-Data Processing for Nearest Neighbor Search. In *International Symposium on Memory Systems*, 2015.

[66] N. Dmitry and S.-S. Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[67] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *ACM conference on Information and knowledge management*, 2008.

[68] D. Ersoz, M. S. Yousif, and C. R. Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *International Conference on Distributed Computing Systems*, 2007.

[69] Q. Fan and Q. Wang. Performance comparison of web servers with different architectures: a case study using high concurrency workload. In *IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[70] D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. *IBM Research Division*, 1994.

[71] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[72] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

[73] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 2009.

[74] B. Furht and A. Escalante. *Handbook of cloud computing*. Springer, 2010.

[75] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases*, 1999.

[76] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[77] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *IEEE/ACM International Symposium on Microarchitecture*, 2017.

[78] F. M. Harper and J. A. Konstan. The Movielens Datasets: History and Context. *ACM Tranactions on Interactive Intelligent Systems*, 2015.

[79] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[80] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[81] E. N. Herness, R. J. High, and J. R. McGee. Websphere Application Server: A foundation for on demand computing. *IBM Systems Journal*, 2004.

[82] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *International Symposium on High Performance Computer Architecture*, 2015.

[83] J. Hu, I. Pyarali, and D. C. Schmidt. Applying the proactor pattern to high-performance web servers. In *International Conference on Parallel and Distributed Computing and Systems*, 1998.

[84] J. C. Hu and D. C. Schmidt. JAWS: A Framework for High-performance Web Servers. In *In Domain-Specific Application Frameworks: Frameworks Experience by Industry*, 1999.

[85] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *ACM Symposium on Theory of Computing*, 1998.

[86] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive Parallelization: Taming Tail Latencies in Web Search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[87] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[88] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling Contention Management from Scheduling. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[89] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005.

[90] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IEEE International Symposium on Workload Characterization*, 2014.

[91] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *ACM Symposium on Cloud Computing*, 2012.

[92] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture*, 2015.

[93] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *ACM International Conference on Web Search and Data Mining*, 2015.

[94] W. Ko, M. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *Parallel and*

Distributed Processing Symposium, 2001.

[95] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *International Conference on Knowledge Discovery and Data Mining*, 2007.

[96] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 2000.

[97] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and thread management. In *Symposium on the Frontiers of Massively Parallel Computing*, 1996.

[98] P.-A. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent mid-tier database caching in SQL server. In *International Conference on Data Engineering*, 2004.

[99] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *International Symposium on Computer Architecture*, 2010.

[100] A. Lesyuk. *Mastering Redmine*. 2013.

[101] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Workshop on Experimental computer science*, 2007.

[102] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM Symposium on Cloud Computing*, 2014.

[103] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[104] Y. Ling, T. Mullen, and X. Lin. Analysis of Optimal Thread Pool Size. *SIGOPS Operating Systems Review*, 2000.

[105] D. Liu and R. Deters. The Reverse C10K Problem for Server-Side Mashups. In *International Conference on Service-Oriented Computing Workshops*, 2008.

[106] P. M. LiVecchi. Performance enhancements for threaded servers, 2004. US Patent 6,823,515.

[107] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *International Symposium on Computer Architecture*, 2014.

[108] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *International Symposium on Computer Architecture*, 2015.

[109] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *International Symposium on High Performance Computer Architecture*, 2016.

[110] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *International Conference on Very Large Data Bases*, 2007.

[111] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. 2010.

[112] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 1993.

[113] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. In *International Symposium on Computer Architecture*, 2011.

[114] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed event-based systems*. 2006.

[115] M. Muja and D. G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[116] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016.

[117] R. M. Needham. Denial of Service. In *ACM Conference on Computer and Communications Security*, 1993.

[118] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, and P. Saab. Scaling Memcache at Facebook. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[119] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, 1999.

[120] D. Pariag, T. Brecht, A. S. Harji, P. A. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. In *European Conference on Computer Systems*, 2007.

[121] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 2016.

[122] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles*, 2017.

[123] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *ACM Symposium on Cloud Computing*, 2015.

[124] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization*, 2011.

[125] D. R. Raymond and S. F. Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 2008.

[126] R. Rojas-Cessa, Y. Kaymak, and Z. Dong. Schemes for fast transmission of flows in data center networks. *IEEE Communications Surveys & Tutorials*, 2015.

[127] D. Schmidt and P. Stephenson. Experience using design patterns to evolve communication software across diverse OS platforms. In *European Conference on Object-Oriented Programming*, 1995.

[128] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 1999.

[129] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *IEEE International Conference on Computer Vision*, 2003.

[130] R. K. Sharma, C. E. Bash, C. D. Patel, R. J. Friedrich, and J. S. Chase. Balance of power: Dynamic thermal management for internet data centers. *IEEE Internet Computing*, 2005.

[131] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 2008.

[132] S. M. Specht and R. B. Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *ISCA International Conference on Parallel and Distributed Computing (and Communications) Systems*, 2004.

[133] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[134] A. Sriraman and T. F. Wenisch. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization*, 2018.

[135] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[136] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[137] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *ACM SIGMOD International Conference on Management of data*, 2009.

[138] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems*, 2010.

[139] K. Terasawa and Y. Tanaka. Spherical LSH for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*, 2007.

[140] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 2010.

[141] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Workshop on Experimental computer science*, 2007.

[142] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[143] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. Timetrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *International Symposium on Microarchitecture*, 2015.

[144] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference*, 2015.

[145] J. R. Von Behren, J. Condit, and E. A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Hot Topics in Operating Systems*, 2003.

[146] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *International Conference on Distributed Computing Systems*, 2017.

[147] Z. Wang and M. F. O'Boyle. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[148] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *ACM Symposium on Operating Systems Principles*, 2001.

[149] W. J. Wilbur and K. Sirotkin. The automatic identification of stop words. *Journal of information science*, 1992.

[150] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM Conference*, 2011.

[151] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference. In *International Symposium on Computer Architecture*, 2016.

RobinHood: Tail Latency-Aware Caching — Dynamically Reallocating from Cache-Rich to Cache-Poor

Daniel S. Berger[1], Benjamin Berg[1], Timothy Zhu[2], Mor Harchol-Balter[1], and Siddhartha Sen[3]

[1]Carnegie Mellon University — [2]Penn State — [3]Microsoft Research

Abstract

Tail latency is of great importance in user-facing web services. However, maintaining low tail latency is challenging, because a single request to a web application server results in multiple queries to complex, diverse backend services (databases, recommender systems, ad systems, etc.). A *request* is not complete until *all of its queries* have completed. We analyze a Microsoft production system and find that backend query latencies vary by more than two orders of magnitude *across backends* and *over time*, resulting in high request tail latencies.

We propose a novel solution for maintaining low request tail latency: repurpose existing caches to mitigate the effects of backend latency variability, rather than just caching popular data. Our solution, RobinHood, dynamically reallocates cache resources from the cache-rich (backends which don't affect request tail latency) to the cache-poor (backends which affect request tail latency). We evaluate RobinHood with production traces on a 50-server cluster with 20 different backend systems. Surprisingly, we find that RobinHood can directly address tail latency even if working sets are much larger than the cache size. In the presence of load spikes, RobinHood meets a 150ms P99 goal 99.7% of the time, whereas the next best policy meets this goal only 70% of the time.

1 Introduction

Request tail latency matters. Providers of large user-facing web services have long faced the challenge of achieving low request latency. Specifically, companies are interested in maintaining low *tail latency*, such as the 99th percentile (P99) of request latencies [26, 27, 36, 44, 63, 83, 92]. Maintaining low tail latencies in real-world systems is especially difficult when incoming *requests* are complex, consisting of multiple *queries* [4, 26, 36, 90], as is common in *multitier architectures*. Figure 1 shows an example of a multitier architecture: each user request is received by an application server, which then sends queries to the necessary *backends*, waits until all queries have completed, and then packages the results for delivery back to the user. Many large web services, such as Wikipedia [71], Amazon [27], Facebook [20], Google [26] and Microsoft, use this design pattern.

The queries generated by a single request are independently processed in parallel, and may be spread over many backend services. Since each request must wait for all of its queries to complete, the overall *request latency* is defined to be the latency of the request's *slowest* query. Even if almost all backends have low tail latencies, the tail latency of the *maximum* of several queries could be high.

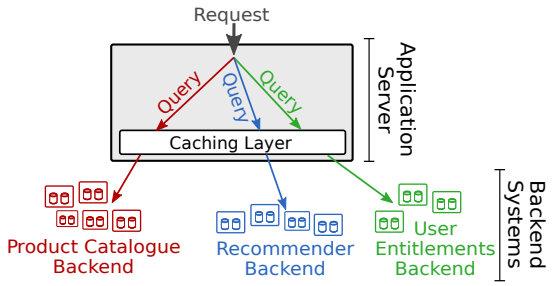


Figure 1: In a multitier system, users submit individual *requests*, which are received by application servers. To complete a request, an application server issues a series of *queries* to various *backend services*. The request is only complete when all of its queries have completed.

For example, consider a stream of requests where each request queries a single backend 10 times in parallel. Each request's latency is equal to the *maximum* of its ten queries, and could therefore greatly exceed the P99 query latency of the backend. The P99 request latency in this case actually depends on a higher percentile of backend query latency [26]. Unfortunately, as the number of backends in the system increases and the workload becomes more heterogeneous, P99 request latency may depend on different (higher or lower) percentiles of query latency for each backend, and determining what these important percentiles are is difficult.

To illustrate this complexity, this paper focuses on a concrete example of a large multitier architecture: the OneRF page rendering framework at Microsoft. OneRF serves a wide range of content including news (msn.com) and online retail software stores (microsoft.com, xbox.com). It relies on more than 20 backend systems, such as product catalogues, recommender systems, and user entitlement systems (Figure 1).

The source of tail latency is dynamic. It is common in multitier architectures that the particular backend causing high request latencies changes over time. For example,

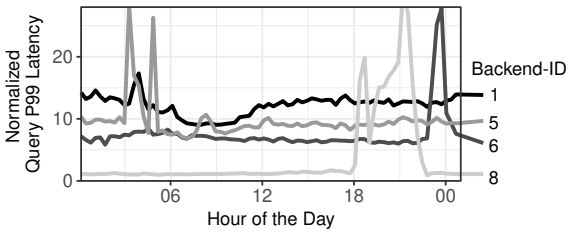


Figure 2: Normalized 99-th percentile (P99) latencies over the course of a typical day for four backends in the OneRF production system. Each backend has the highest tail latency among all backends at some point during the day, indicating that latencies are not only unbalanced between backends, but the imbalance changes over time.

Figure 2 shows the P99 query latency in four typical backend services from OneRF over the course of a typical day. Each of the four backends at some point experiences high P99 query latency, and is thus responsible for some high-latency requests. However, this point happens at a different time for each backend. Thus any mechanism for identifying the backends that affect tail request latency should be dynamic—accounting for the fact that the latency profile of each backend changes over time.

Existing approaches. Some existing approaches for reducing tail latency rely on load balancing between servers and aim to equalize query tail latencies between servers. This should reduce the maximum latency across multiple queries. Unfortunately, the freedom to load balance is heavily constrained in a multitier architecture, where a given backend typically is unable to answer a query originally intended for another backend system (e.g., the user entitlements backend cannot answer product catalogue queries). While some limited load balancing can be done between replicas of a single backend system, load balancing is impossible across *different* backends.

Alternatively, one might consider reducing tail latency by dynamically auto-scaling backend systems—temporarily allocating additional servers to the backends currently experiencing high latency. Given the rapid changes in latency shown in Figure 2, it is important to be able to scale backends quickly. Unfortunately, dynamic auto-scaling is difficult to do quickly in multitier systems like OneRF because backends are stateful [30]. In fact, many of the backends at Microsoft do some form of auto-scaling, and Figure 2 shows that the systems are still affected by latency spikes.

The RobinHood solution. In light of these challenges, we suggest a novel idea for minimizing request tail latency that is agnostic to the design and functionality of the backend services. We propose repurposing the existing caching layer (see Figure 1) in the multitier system to directly address request tail latency by dynamically partitioning the cache. Our solution, *RobinHood*, dynamically

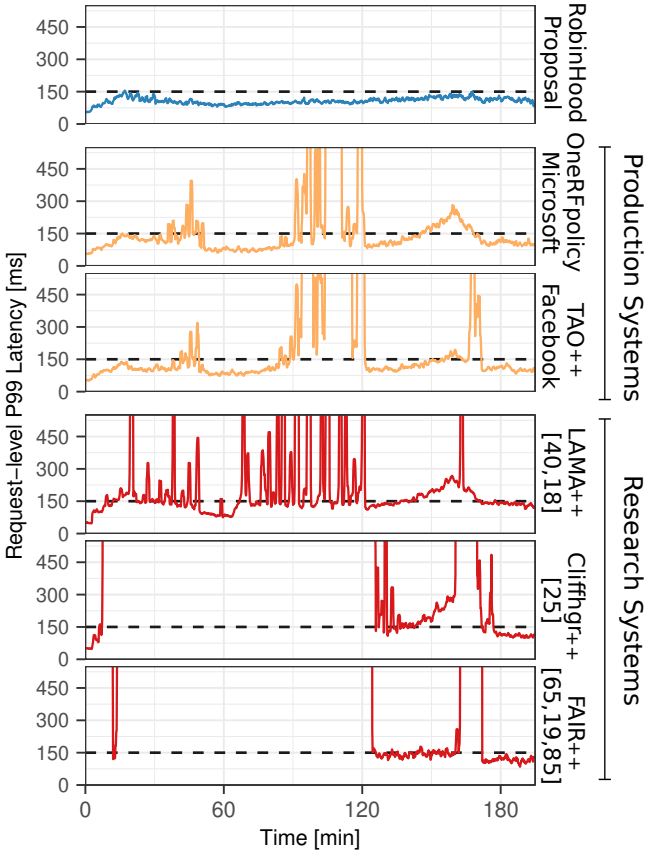


Figure 3: Comparison of the P99 request latency of RobinHood, two production caching systems, and three state-of-the-art research caching systems, which we emulated in our testbed. All systems are subjected to three load spikes, as in Figure 2. We draw a dashed line at 150ms, which is the worst latency under RobinHood.

allocates cache space to those backends responsible for high request tail latency (cache-poor backends), while stealing space from backends that do not affect the request tail latency (cache-rich backends). In doing so, RobinHood makes compromises that may seem counter-intuitive (e.g., significantly increasing the tail latencies of certain backends) but which ultimately improve overall request tail latency. Since many multitier systems already incorporate a caching layer that is capable of dynamic partitioning, RobinHood can be deployed with very little additional overhead or complexity.

RobinHood is not a traditional caching system. While many multitier systems employ a caching layer, these caches are often designed only to improve *average* (not tail) latency of *individual queries* (not requests) [3, 11, 17, 25, 40]. In all of the production workloads we study, an application's working set is larger than the available cache space, and thus the caching layer can improve average query latency by allowing fast accesses (cache hits) to the most popular data. By contrast, request tail latency is

caused almost entirely by cache misses. In fact, conventional wisdom says that when the application's working set does not fit entirely in the cache, the caching layer does not directly address tail latency [26]. Thus, despite various efforts to optimize the caching layer in both industry and academia (see Section 7), none of these systems are designed to reduce request tail latency.

Contributions. RobinHood is the first caching system that minimizes the request tail latency. RobinHood is driven by a lightweight cache controller that leverages existing caching infrastructure and is agnostic to the design and functionality of the backend systems.

We implement[1] and extensively evaluate the RobinHood system along with several research and production caching systems. Our 50-server testbed includes 20 backend systems that are modeled after the 20 most queried backends from Microsoft's OneRF production system. Figure 3 shows a preview of an experiment where we mimic the backend latency spikes in OneRF: RobinHood meets a 150ms P99 goal 99.7% of the time, whereas the next best policy meets this goal only 70% of the time.

Our contributions are the following:

- **Section 2.** We find that there are many different types of requests, each with their own *request structure* that defines which backend systems are queried. We analyze structured requests within the OneRF production system, and conclude that request structure must be incorporated by any caching system seeking to minimize request tail latency.
- **Section 3.** We present RobinHood, a dynamic caching system which aims to minimize request tail latency by considering request structure. RobinHood identifies which backends contribute to the tail over time, using a novel metric called request blocking count (RBC).
- **Section 4.** We implement RobinHood as a scalable distributed system. We also implement the first distributed versions of state-of-the-art research systems: LAMA [40], Cliffhanger [25], and FAIR [19, 65, 85] to use for comparison.
- **Section 5.** We evaluate RobinHood and prior systems against simultaneous latency spikes across multiple backends, and show that RobinHood is far more robust while imposing negligible overhead.

We discuss how to generalize RobinHood to architecture beyond OneRF in **Section 6**, survey the related work in **Section 7**, and conclude in **Section 8**.

2 Background and Challenges

The RobinHood caching system targets tail latency in multitier architectures, where requests depend on queries to many backends. One such system, the OneRF system, serves several Microsoft storefront properties and relies on a variety of backend systems. Each OneRF application server has a *local* cache. Incoming requests are split into queries which first lookup up in the cache. Cache misses are then forwarded, in parallel, to clusters of backend servers. Once each query has been answered, the application server can serve the user request. Thus, each request takes as long as its slowest query. A OneRF request can send any number of queries (including 0) to each backend system.

Before we describe the RobinHood algorithm in Section 3, we discuss the goal of RobinHood in more depth, and how prior caching systems fail to achieve this goal.

2.1 The goal of RobinHood

The key idea behind RobinHood is to identify backends whose queries are responsible for high P99 request latency, which we call "cache-poor" backends. RobinHood then shifts cache resources from the other "cache-rich" backends to the cache-poor backends. RobinHood is a departure from "fair" caching approaches [65], treating queries to cache-rich backends unfairly as they do not affect the P99 request latency. For example, increasing the latency of a query that occurs in parallel with another, longer query, will not increase the *request* latency. By sacrificing the performance of cache-rich backends, RobinHood frees up cache space.

RobinHood allocates free cache space to cache-poor backends (see Section 3). Additional cache space typically improves the hit ratios of these backends, as the working sets of web workloads do not fit into most caches [3, 20, 41, 61, 72]. As the hit ratio increases, fewer queries are sent to the cache-poor backends. Since backend query latency is highly variable in practice [4, 26, 29, 36, 45, 62, 68, 83], decreasing the number of queries to a backend will decrease the number of high-latency queries observed. This will in turn improve the P99 request latency.

In addition, sending fewer queries can also reduce resource congestion and competition in the backends, which is often the cause of high tail latency [26, 35, 77]. Small reductions in resource congestion can have an outsized impact on backend latency [34, 39] and thus significantly improve the request P99 (as we will see in Section 5).

2.2 Challenges of caching for tail latency

We analyze OneRF traces collected over a 24 hour period in March 2018 in a datacenter on the US east coast. The traces contain requests, their queries, and the query latencies for the 20 most queried backend systems, which account for more than 99.95% of all queries.

We identify three key obstacles in using a caching system to minimizing tail request latencies.

[1] RobinHood's source code is available at `https://github.com/dasebe/robinhoodcache`.

2.2.1 Time-varying latency imbalance

As shown in Figure 2, it is common for the latencies of different backends to vary widely. Figure 5 shows that the latency across the 20 backends varies by more than 60×. The fundamental reason for this latency imbalance is that several of these backend systems are complex, distributed systems in their own right. They serve multiple customers within the company, not just OneRF.

In addition to high latency imbalance, backend latencies also change over time (see Figure 2). These changes are frequently caused by customers other than OneRF and thus occur independently of the request stream seen by OneRF applications servers.

Why latency imbalance is challenging for existing systems. Most existing caching systems implicitly assume that latency is balanced. They focus on optimizing cache-centric metrics (e.g., hit ratio), which can be a poor representation of overall performance if latency is imbalanced. For example, a common approach is to partition the cache in order to provide fairness guarantees about the hit ratios of queries to different backends [19, 65, 85]. This approach is represented by the FAIR policy in Figure 3, which dynamically partitions the cache to equalize backend cache hit ratios. If latencies are imbalances between the backends, two cache misses to different backends should not be treated equally. FAIR fails to explicitly account for the latency of cache misses and thus may result in high request latency.

Some production systems do use latency-aware static cache allocations, e.g., the "arenas" in Facebook's TAO [20]. However, manually deriving the optimal static allocation is an open problem [20], and even an "optimal" static allocation will become stale as backend latencies vary over time (see Section 5).

2.2.2 Latency is not correlated with specific queries nor with query rate

We find that high latency is not correlated with specific queries as assumed by cost-aware replacement policies [21, 52]. Query latency is also not correlated with a query's popularity (the rate at which the query occurs), but rather reflects a more holistic state of the backend system at some point in time. This is shown in Figure 4 with a scatterplot of a query's popularity and its latency for the four OneRF backends shown in Figure 2 (other backends look similar).

We also find that query latency is typically not correlated with a backend's query rate. For example, the seventh most queried backend receives only about 0.06× as many queries as the most backend, but has 3× the query latency (Figure 5). This is due to the fact that backend systems are used by customers other than OneRF. Even if OneRF's query rate to a backend is low, another service's query stream may be causing high backend latency. Additionally, queries to some backends take inherently

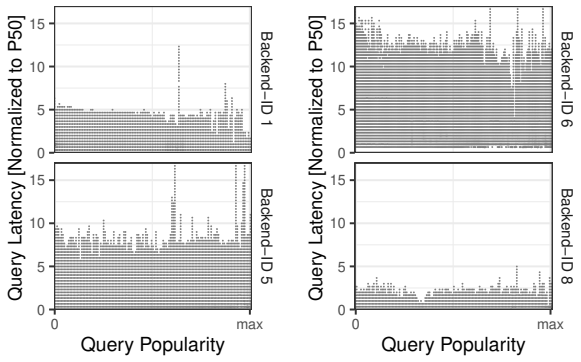


Figure 4: Scatterplots of query popularity and query latency for each backend from Figure 2. We find that query latency is neither correlated with query rate nor with particular queries.

longer than others, e.g., generating a personalized product recommendation takes 4× longer than looking up a catalogue entry.

Why uncorrelated latency is challenging for existing systems. Many caching schemes (including OneRF) share cache space among the backends and use a common eviction policy (such as LRU). Shared caching systems [17, 52] inherently favor backends with higher query rates [9]. Intuitively, this occurs because backends with higher query rates have more opportunities for their objects to be admitted into the cache. Cost-aware replacement policies also suffer from this problem, and are generally ineffective in multitier architectures such OneRF as their assumptions (high latency is correlated with specific queries) are not met.

Another common approach is to partition cache space to maximize overall cache hit ratios as in Cliffhanger [25]. All these approaches allocate cache space in proportion to query rate, which leads to suboptimal cache space allocations when latency is uncorrelated with query rate. As shown in Figure 3, both OneRF and Cliffhanger lead to high P99 request latency. In order to minimize request tail latency, a successful caching policy must directly incorporate backend latency, not just backend query rates.

2.2.3 Latency depends on request structure, which varies greatly

The manner in which an incoming request is split into parallel backend queries by the application server varies between requests. We call the mapping of a request to its component backend queries the *request structure*.

To characterize the request structure, we define the number of parallel queries to a single backend as the backend's *batch size*. We define the number of distinct backends queried by a request as its *fanout*. For a given backend, we measure the average batch size and fanout of requests which reference this backend.

Table 1 summarizes how the query traffic of different

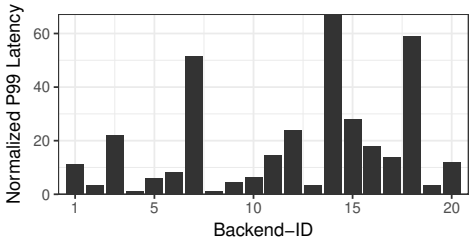


Figure 5: Normalized P99 latencies for the 20 most queried backends in the OneRF system during a typical 10 minute period. The backends are ordered by their query rates during this period. We see that query rate is not correlated with backend tail latency.

backends is affected by the request structure. We list the percentage of the overall number of queries that go to each backend, and the percentage of requests which reference each backend. We also list the average batch size and fanout by backend. We can see that all of these metrics vary across the different backends and are not strongly correlated with each other.

Why request structure poses a challenge for existing systems. There are few caching systems that incorporate latency into their decisions, and they consider the average query latency as opposed to the tail request latency [18, 40]. We find that even after changing these *latency aware* systems to measure the P99 query latency, they remain ineffective (see LAMA++ in Figure 3).

These systems fail because a backend with high query latency does not always cause high request latency. A simple example would be high query latency in backend 14. As backend 14 occurs in less than 0.2% of all requests, its impact on the P99 request latency is limited—even if backend 14 was arbitrarily slow, it could not be responsible for *all* of the requests above the P99 request latency. A scheme that incorporates query rate and latency might decide to allocate most of the cache space towards backend 14, which would not improve the P99 request latency.

While the specific case of backend 14 might be simple to detect, differences in batch sizes and fanout give rise to complicated scenarios. For example, Figure 5 shows that backend 3's query latency is higher than backend 4's query latency. Table 1 shows that, while backend 3 has a large batch size, backend 4 occurs in 4.5× more requests, which makes backend 4 more likely to affect the P99 request latency. In addition, backend 4 occurs in requests with a 55% smaller fanout, which makes it more likely to be the slowest backend, whereas backend 3's query latency is frequently hidden by slow queries to other backends.

As a consequence, minimizing request tail latency is difficult unless request structure is explicitly considered.

Backend-ID	Query %	Request %	Mean Batch Size	Mean Fanout
1	37.7%	14.7%	15.4	5.6
2	16.0%	4.5%	32.3	7.4
3	15.3%	4.5%	25.7	7.4
4	14.0%	20.0%	1.6	4.8
5	7.7%	19.0%	1.9	4.9
6	4.2%	4.7%	14.5	7.3
7	2.4%	10.8%	2.0	5.3
8	1.6%	15.5%	1.0	5.3
9	0.7%	3.4%	2.0	7.5
10	0.2%	0.7%	2.5	9.1

Table 1: Four key metrics describing the 10 most queried OneRF backends. Backend-IDs are ordered by query rate starting with the most queried backend, backend 1. Query % describes the percentage of the total number of queries directed to a given backend. Request % denotes the percentage of requests with at least one query to the given backend. Batch size describes the average number of parallel queries made to the given backend across requests with at least one query to that backend. Fanout describes the average number of backends queried across requests with at least one query to the given backend.

3 The RobinHood Caching System

In this section, we describe the basic RobinHood algorithm (Section 3.1), how we accommodate real-world constraints (Section 3.2), and the high-level architecture of RobinHood (Section 3.3). Implementation details are discussed in Section 4.

3.1 The basic RobinHood algorithm

To reallocate cache space, RobinHood repeatedly taxes every backend by reclaiming 1% of its cache space, identifies which backends are cache-poor, and redistributes wealth to these cache-poor backends.

RobinHood operates over time windows of Δ seconds, where $\Delta = 5$ seconds in our implementation.[2] Within a time window, RobinHood tracks the latency of each request. Since the goal is to minimize the P99 request latency, RobinHood focuses on the set of requests, S, whose request latency is between the P98.5 and P99.5 (we explain this choice of range below). For each request in S, RobinHood tracks the ID of the backend corresponding to the slowest query in the request. RobinHood then counts the number of times each backend produced the slowest query in a request. We call each backend's total count its *request blocking count* (RBC). Backends with a high RBC are frequently the bottleneck in slow requests. RobinHood thus considers a backend's RBC as a measure of how cache-poor it is, and distributes the pooled tax to each backend in proportion to its RBC.

Choosing the RBC metric. The RBC metric captures key aspects of request structure. Recall that, when minimizing request tail latency, it is not sufficient to know

[2] This parameter choice is determined by the time it takes to reallocate 1% of the cache space in off-the-shelf caching systems; see Section 4.

only that a backend produces high query latencies. This backend must also be queried in such a way that it is frequently the slowest backend queried by the slowest requests in the system. Metrics such as batch size and fanout width will determine whether or not a particular backend's latencies are hidden or amplified by the request structure. For example, if a slow backend is queried in parallel with many queries to other backends (high fanout width), the probability of the slow backend producing the slowest query may be relatively small. We would expect this to result in a lower RBC for the slow backend than its query latency might suggest. A backend with a high RBC indicates not only that the backend produced high-latency queries, but that reducing the latency of queries to this backend would have actually reduced the latency of the requests affecting the P99 request latency.

Choosing the set S. The set S is chosen to contain requests whose latency is close to the P99 latency, specifically between the P98.5 and P99.5 latencies. Alternatively, one might consider choosing S to be the set requests with latencies greater than or equal to the P99. However, this set is known to include extreme outliers [27] whose latency, even if reduced significantly, would still be greater than the P99 latency. Improving the latency of such outliers would thus be unlikely to change the P99 request latency. Our experiments indicate that choosing a small interval around the P99 filters out most of these outliers and produces more robust results.

3.2 Refining the RobinHood algorithm

The basic RobinHood algorithm, described above, is designed to directly address the key challenges outlined in Section 2. However, real systems introduce additional complexity that must be addressed. We now describe two additional issues that must be considered to make the RobinHood algorithm effective in practice.

Backends appreciate the loot differently. The basic RobinHood algorithm assumes that redistributed cache space is filled immediately by each backend's queries. In reality, some backends are slow to make use of the additional cache space because their hit ratios are already high. RobinHood detects this phenomenon by monitoring the gap between the allocated and the used cache capacity for each backend. If this gap is more than 30% of the used cache space, RobinHood temporarily ignores the RBC of this backend to avoid wasting cache space. Note that such a backend may continue to affect the request tail latency. RobinHood instead chooses to focus on backends which can make use of additional cache space.

Local decision making and distributed controllers. The basic RobinHood algorithm assumes an abstraction of a single cache with one partition per backend. In reality (e.g., at OneRF), incoming requests are load balanced across a cluster of application servers, each of which has its own local cache (see Section 2). Due to random load balancing, two otherwise identical partitions on different application servers may result in different hit ratios[3]. Therefore, additional cache space will be consumed at different rates not only per backend, but also per application server. To account for this, RobinHood's allocation decisions (such as imposing the 30% limit described above) are made locally on each application server. This leads to a distributed controller design, described in Section 3.3.

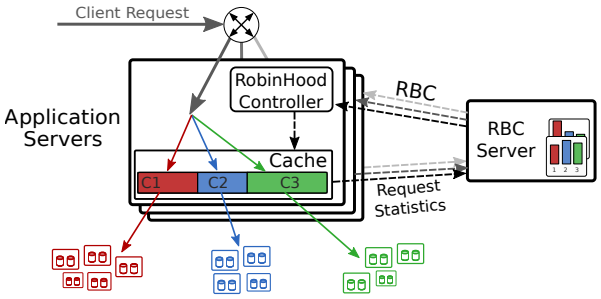


Figure 6: RobinHood adds a distributed controller to each application server and a latency statistics (RBC) server.

One might worry that the choice of distributed controllers could lead to diverging allocations and cache space fragmentation across application servers over time. However, as long as two controllers exchange RBC data (see Section 3.3), their cache allocations will quickly converge to the same allocation regardless of initial differences between their allocations. Specifically, given $\Delta = 5$ seconds, any RobinHood cache (e.g., a newly started one) will converge to the average allocation within 30 minutes assuming all servers see sufficient traffic to fill the caches. In other words, the RobinHood cache allocations are *not* in danger of "falling off a cliff" due to diverging allocations — RobinHood's distributed controllers will always push the caches back to the intended allocation within a short time span.

3.3 RobinHood architecture

Figure 6 shows the RobinHood architecture. It consists of application servers and their caches, backend services, and an RBC server.

RobinHood requires a caching system that can be dynamically resized. We use off-the-shelf memcached instances to form the caching layer on each application server in our testbed (see Section 4). Implementing RobinHood requires two additional components not currently used by production systems such as OneRF. First, we add a lightweight cache controller to each application server. The controller implements the RobinHood algorithm (Sections 3.1 and 3.2) and issues resize commands to the local cache's partitions. The input for each controller is the RBC, described in Section 3.1. To prevent

[3] While most caches will contain roughly the same data, it is likely that at least one cache will look notably different from the others.

an all-to-all communication pattern between controllers, we add a centralized RBC server. The RBC server aggregates request latencies from all application servers and computes the RBC for each backend. In our implementation, we modify the application server caching library to send (in batches, every second) each request's latency and the backend ID from the request's longest query. In the OneRF production system, the real-time logging framework already includes all the metrics required to calculate the RBC, so RobinHood does not need to change application libraries. This information already exists in other production systems as well, such as at Facebook [77]. The controllers poll the RBC server for the most recent RBCs each time they run the RobinHood algorithm.

Fault tolerance and scalability. The RobinHood system is robust, lightweight, and scalable. RobinHood controllers are distributed and do not share any state, and RBC servers store only soft state (aggregated RBC from the last one million requests, in a ring buffer). Both components can thus quickly recover after a restart or crash. Just as RobinHood can recover from divergence between cache instances due to randomness (see Section 3.2), RobinHood will recover from any measurement errors that might result in bad reallocation messages being sent to the controllers. The additional components required to run RobinHood (controller and statistics server) are not on the critical path of requests and queries, and thus do not impose any latency overhead. RobinHood imposes negligible overhead and can thus scale to several hundred application servers (Section 5.6).

4 System Implementation and Challenges

To demonstrate the effectiveness and deployability of RobinHood, we implement the RobinHood architecture using an off-the-shelf caching system. In addition, we implement five state-of-the-art caching systems (further described in Section 5.1) on top of this architecture.

4.1 Implementation and testbed

The RobinHood controller is a lightweight Python process that receives RBC information from the global RBC server, computes the desired cache partition sizes, and then issues resize commands to the caching layer. The RBC server and application servers are highly concurrent and implemented in Go. The caching layer is composed of off-the-shelf memcached instances, capable of dynamic resizing via the memcached API. Each application server has a local cache with 32 GB cache capacity.

To test these components, we further implement different types of backend systems and a concurrent traffic generator that sends requests to the application servers. On average, a request to the application server spawns 50 queries. A query is first looked up in the local memcached instance; cache misses are then forwarded to the corresponding backend system. During our experiments the average query rate of the system is 200,000 queries per second (over 500,000 peak). To accommodate this load we had to build highly scalable backend systems. Specifically, we use three different types of backends. A distributed key-value store that performs simple lookup queries (similar to OneRF's product rating and query service). A fast MySQL cluster performs an indexed-join and retrieves data from several columns (similar to OneRF's product catalogue systems). And, a custom-made matrix-multiplier system that imitates a recommendation prediction (similar to various OneRF recommender backends).

Our experimental testbed consists of 16 application servers and 34 backend servers divided among 20 backend services. These components are deployed across 50 Microsoft Azure D16 v3 VMs[4].

4.2 Implementation challenges

The central challenge in implementing our testbed was scaling our system to handle 200,000-500,000 queries per second across 20 different backend systems.

For example, our initial system configuration used a sharded distributed caching layer. We moved away from this design because the large batch size within some requests (up to 300 queries) meant that every cache had to be accessed [20, 77]. Our current testbed matches the design used in the OneRF production system in that each application server only queries its local cache.

Another challenge we compensate for is the delay of reallocating cache space in off-the-shelf memcached instances. Memcached's reallocation API works at the granularity of 1MB pages. To reallocate 1% of the cache space (Section 3), up to 327 memcached pages need to be reallocated. To reallocate a page, memcached must acquire several locks, in order to safely evict page contents. High load in our experiments leads to memcached-internal lock contention, which delays reallocation steps. Typically (95% of the time), reallocations take no longer than 5 seconds, which is why $\Delta = 5$ seconds (in Section 3). To tolerate atypical reallocations that take longer than 5 seconds, the RobinHood controller can defer cache allocations to future iterations of the RobinHood algorithm.

Finally, we carefully designed our testbed for reproducible performance results. For example, to deal with complex state throughout the deployment (e.g., in various backends), we wipe all state between repeated experiments, at the cost of a longer warmup period.

4.3 Generating experimental data

Microsoft shared with us detailed statistics of production traffic in the OneRF system for several days in 2017 and 2018 (see Section 2). We base our evaluation on the

[4]The servers are provisioned with 2.4 GHz Intel E5-2673v3 with eight cores, 64GB memory, 400GB SSDs with up to 24000 IOPS, and 8Gbit/s network bandwidth.

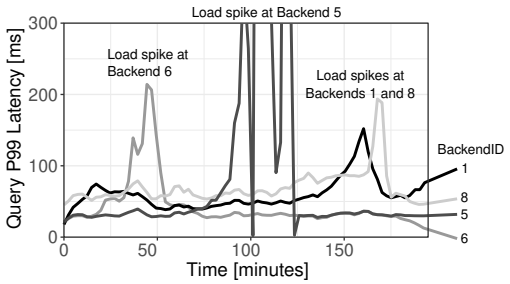


Figure 7: P99 latency of backend queries in our experiment. The four latency spikes emulate the latency spikes in the OneRF production systems (see Figure 2).

2018 dataset. The dataset describes queries to more than 40 distinct backend systems.

In our testbed, we replicate the 20 most queried backend systems, which make up more than 99.95% of all queries. Our backends contain objects sampled from the OneRF object size distribution. Across all backends, object sizes range between a few bytes to a few hundred KB, with a mean of 23 KB. In addition, our backends approximately match the design of the corresponding OneRF backend.

Our request traffic replicates key features of production traffic, such as an abundance of several hundreds of different request types, each with their own request structures (e.g., batch size, fanout, etc). We sample from the production request type distribution and create four-hour-long traces with over 50 million requests and 2.5 billion queries. We verified that our traces preserve statistical correlations and locality characteristics from the production request stream. We also verified that we accurately reproduce the highly varying cacheability of different backend types. For example, the hit ratios of the four backends with latency spikes (Figure 2) range between 81-92% (backend 1), 51-63% (backend 5), 37-44% (backend 6), and 96-98% (backend 8) in our experiments. The lowest hit ratio across all backends is 10% and the highest is 98%, which means that the P99 tail latency is always composed of cache misses (this matches our observations from the production system). None of the working sets fit into the application server's cache, preventing trivial scenarios as mentioned in the literature [26].

5 Evaluation

Our empirical evaluation of RobinHood focuses on five key questions. Throughout this section, our goal is to meet a P99 request latency Service Level Objective (SLO) of 150ms, which is a typical goal for user-facing web applications [26, 27, 34, 50, 56, 59, 90, 91]. Every 5s, we measure the P99 over the previous 60s. We define the SLO violation percentage to be the fraction of observations where the P99 does not meet the SLO. We compare

Name	Optimization goal	Dynamic	Latency-aware	Request structure
RobinHood	Minimize *request* P99	yes	yes	yes
OneRFpolicy	Minimize miss ratio	no	no	no
TAO$_{++}$ [20]	Minimize *request* P99	no	no	no
Cliffhgr$_{++}$ [25]	Minimize miss ratio	yes	no	no
FAIR$_{++}$ [19,65]	Equalize miss ratios	yes	no	no
LAMA$_{++}$ [18,40]	Equalize *query* P99	yes	yes	no

Table 2: The six caching systems evaluated in our experiments. RobinHood is the only dynamic caching system that seeks to minimize the tail *request* latency and the first caching system that utilizes request structure rather than just queries.

RobinHood to five state-of-the-art caching systems, defined in Section 5.1, and answer the following questions:

Section 5.2: How much does RobinHood improve SLO violations for OneRF's workload? Quick answer: RobinHood brings SLO violations down to 0.3%, compared to 30% SLO violations under the next best policy.

Section 5.3: How much variability can RobinHood handle? Quick answer: for quickly increasing backend load imbalances, RobinHood maintains SLO violations below 1.5%, compared to 38% SLO violations under the next best policy.

Section 5.4: How robust is RobinHood to simultaneous latency spikes? Quick answer: RobinHood maintains less than 5% SLO violations, while other policies do significantly worse.

Section 5.5: How much space does RobinHood save? Quick answer: The best clairvoyant static allocation requires 73% more cache space in order to provide each backend with its maximum allocation under RobinHood.

Section 5.6: What is the overhead of running RobinHood? Quick answer: RobinHood introduces negligible overhead on network, CPU, and memory usage.

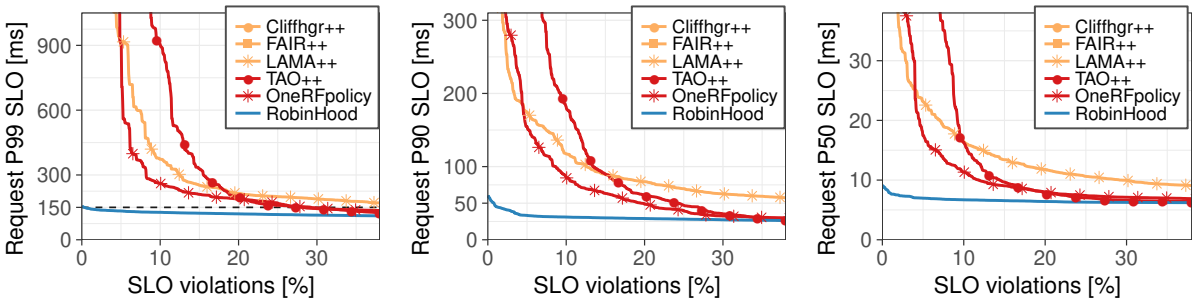
5.1 Competing caching systems

We compare RobinHood to two production systems and three research caching systems listed in Table 2.

The two production systems do not currently dynamically adjust the caches. OneRFpolicy uses a single shared cache, which matches the configuration used in the OneRF production system. TAO$_{++}$ uses static allocations. As manually deriving the optimal allocation is an open problem [20], we actually use RobinHood to find a good allocation for the first 20% of the experiment in Section 5.2. TAO$_{++}$ then keeps this allocation fixed throughout the experiment. This is an optimistic version of TAO (thus the name TAO$_{++}$) as finding its allocation would have been infeasible without RobinHood.[5].

We evaluate three research systems, Cliffhanger [25], FAIR [19,65,85], and LAMA [40] (which is conceptually

[5]For example , we have also experimented with brute-force searches, but the combinatorial search space for 20 partitions is too large. We did not find a better allocation over the course of 48 hours.



(a) P99 SLOs (b) P90 SLOs (c) P50 SLOs

Figure 8: Request SLO as a function of SLO violations for (a) P99 SLOs, (b) P90 SLOs, (c) P50 SLOs. For a given violation percentage the plot shows what SLO would have been violated with that frequency. A lower value indicates a system is able to meet lower latency SLOs with fewer SLO violations. RobinHood is the only system that is robust against latency spikes on backends and violates a 150ms P99 SLO only 0.3% of the time (dashed horizontal line in (a)). FAIR_{++} and Cliffhgr_{++} are not shown as their SLO violations are too high to be visible.

similar to [18]). All three systems dynamically adjust the cache, but required major revisions before we could compare against them. All three research systems are only designed to work on a single cache. Two of the systems, Cliffhanger and FAIR, are not aware of multiple backends, which is typical for application-layer caching systems. They do not incorporate request latency or even query latency, as their goal is to maximize the overall cache hit ratio and the fairness between users, respectively. We adapt Cliffhanger and FAIR to work across distributed application servers by building a centralized statistics server that aggregates and distributes their measurements. We call their improved versions Cliffhgr_{++} and FAIR_{++}. LAMA's goal is to minimize mean query latency, not tail query latency (and it does not consider request latency). To make LAMA competitive, we change the algorithm to use P99 query latency and a centralized statistics server. We call this improved version LAMA_{++}.

Our evaluation does not include cost-aware replacement policies for shared caches, such as Greedy-Dual [21] or GD-Wheel [52]. Due to their high complexity, it is challenging to implement them in concurrent caching systems [11]; we are not aware of any production system that implements these policies. Moreover, the OneRF workload does not meet the basic premise of cost-aware caching (Section 2.2.2).

5.2 How much does RobinHood improve SLO violations for OneRF's workload?

To compare RobinHood to the five caching systems above, we examine a scenario that replicates the magnitude and rate with which query latency varies over time in the OneRF production system, as shown in Figure 2. In production systems, this variability is often caused by temporary spikes in the traffic streams of other services which share these backends (Section 2).

Experimental setup. To make experiments reproducible, we emulate latency imbalances by imposing a variable

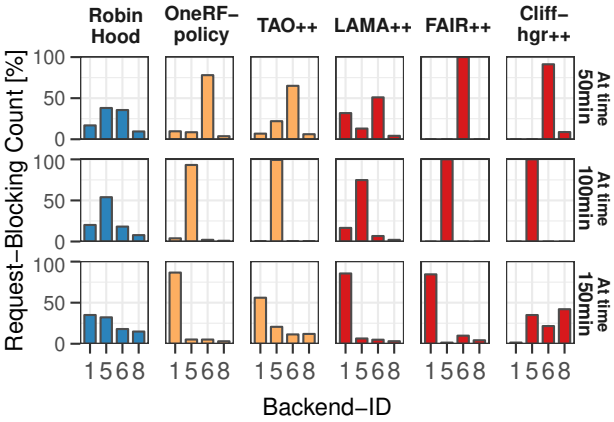


Figure 9: Comparison of how well different caching systems balance the RBC at three times in the experiment from Figure 7. RobinHood is the only system whose RBCs do not significantly exceed 50%.

resource limit on several backends in our testbed. Depending on the backend type (see Section 4), a backend is either IO-bound or CPU-bound. We use Linux control groups to limit the available number of IOPS or the CPU quota on the respective backends. For example, to emulate the latency spike on Backend 6 at 4AM (Figure 2), we limit the number of IOPS that this backend is allowed to perform, which mimics the effect of other traffic streams consuming these IOPS.

We emulate latency variability across the same four backends as shown in Figure 2: backends 1, 5, 6, and 8. Our experiments span four hours each, and we use the first 25% of the experiment time to warm the backends and caches. Figure 7 shows the P99 latency of queries in our experiments under the OneRFpolicy (ignoring an initial warmup period). We verified that the latency spikes are similar in magnitude to those we observe in the OneRF production system (Figure 2).

Experimental results. We compare RobinHood to the

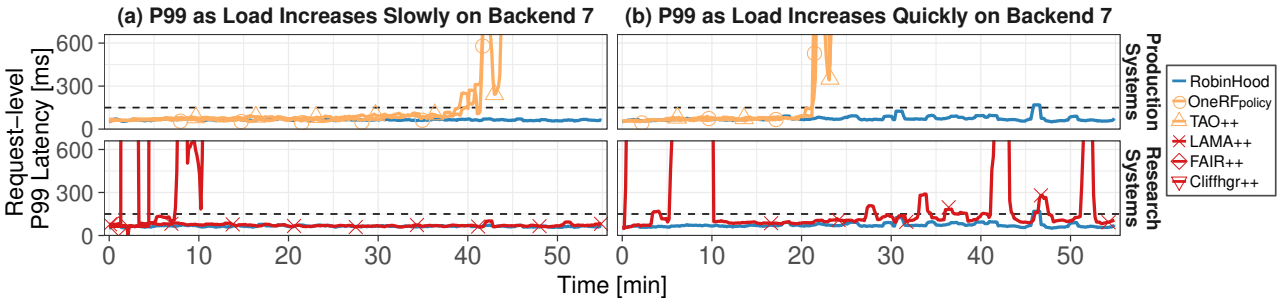


Figure 10: Results from sensitivity experiments where *latency is uncorrelated with query rate*. The load on Backend 7 increases either slowly (left column) or quickly (right column). Even when load increases quickly, RobinHood violates a 150ms P99 SLO less than 1.5% of the time. In contrast, the second best system (LAMA$_{++}$) has 38% SLO violations.

competing systems along several dimensions: the P99 request latency, the rate of SLO violations, and how well they balance the RBC between backends.

Figure 3 shows the P99 request latency for each system over the course of the experiment. Throughout the experiment, RobinHood maintains a P99 below our SLO target of 150ms. Both the production and research systems experience high P99 latencies during various latency spikes, and Cliffhgr$_{++}$ and FAIR$_{++}$ even experience prolonged periods of instability. RobinHood improves the P99 over every competitor by at least 3x during some latency spike.

Figure 8 summarizes the frequency of SLO violations under different SLOs for each caching system in terms of the P99, P90 and P50 request latency. If the goal is to satisfy the P99 SLO 90% of the time, then the graph indicates the strictest latency SLO supported by each system (imagine a vertical line at 10% SLO violations). If the goal is to meet a particular P99 SLO such as 150ms, then the graph indicates the fraction of SLO violations (imagine a horizontal line at 150ms). The figure does not show FAIR$_{++}$ and Cliffhgr$_{++}$ as the percentage of SLO violations is too high to be seen. We find that RobinHood can meet much lower latency SLOs than competitors with almost no SLO violations. For example, RobinHood violates a P99 SLO of 150ms (Figure 8a) less than 0.3% of the time. By contrast, the next best policy, OneRFpolicy, violates this same SLO 30% of the time.

Throughout these experiments, RobinHood targets the P99 request latency (Section 3). We discuss in Section 6 how to generalize RobinHood's optimization goal. However, even though RobinHood focuses on the P99, it still performs well on the P90 and P50. For example, for a P90 SLO of 50ms (Figure 8b), RobinHood leads to less than 1% SLO violations, whereas OneRFpolicy leads to about 20% SLO violations.

Figure 9 shows the RBC (defined in Section 3) for the four backends affected by latency spikes under each caching system at 50 min, 100 min, and 150 min into the experiment, respectively. This figure allows us to quantify how well each system uses the cache to balance RBCs across backend systems. We refer to the *dominant backend* as the backend which accounts for the highest percentage of RBCs. RobinHood achieves the goal of maintaining a fairly even RBC balance between backends—in the worst case, RobinHood allows the dominant backend to account for 54% of the RBC. No other competitor is able to keep the dominating backend below 85% in all cases and even the average RBC of the dominant backend exceeds 70%.

5.3 How much variability can RobinHood handle?

To understand the sensitivity of each caching policy to changes in backend load, we perform a more controlled sensitivity analysis.

Experimental setup. To emulate the scenario that some background work is utilizing the resources of a backend, we limit the resources available to a backend system. In these experiments, we continuously decrease the resource limit on a single backend over a duration of 50 minutes. We separately examine two backends (backend 1 and backend 7) and test two different rates for the resource decrease—the "quick" decrease matches the speed of the fastest latency spikes in the OneRF production system, and the "slow" decrease is about one third of that speed.

Experimental results. Figure 10 shows the P99 request latency under increasing load on backend 7. This experiment benchmarks the typical case where *high latency is uncorrelated with query rate* (Section 2). Figure 10(a) shows that, when load increases slowly, RobinHood never violates a 150ms SLO. In contrast, OneRFpolicy and TAO$_{++}$ are consistently above 150ms after 40min, when the latency imbalance becomes more severe than in Section 5.2. Of the research systems, FAIR$_{++}$ and Cliffhgr$_{++}$ are above 150ms after 10min. LAMA$_{++}$, the only system that is latency aware, violates the SLO 3.3% of the time.

Figure 10(b) shows that, when load increases quickly, RobinHood maintains less than 1.5% SLO violations. All other systems become much worse, e.g., OneRFpolicy and TAO$_{++}$ are above 150ms already after 20min. The second best system, LAMA$_{++}$, violates the SLO more

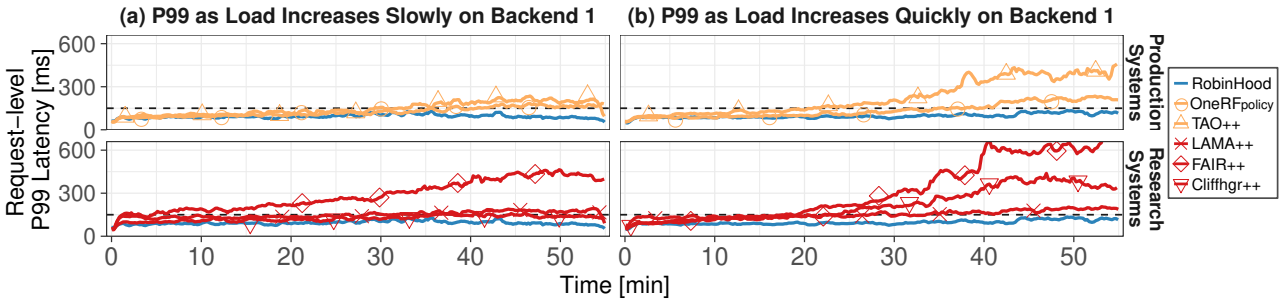


Figure 11: Results from sensitivity experiments where *latency is correlated with query rate*. The load on Backend 1 increases either slowly (left column) or quickly (right column). Even when load increases quickly, RobinHood never violates a 150ms P99 SLO. In contrast, the second best system (OneRFpolicy) has 33% SLO violations.

than 38% of the time, which is 25× more frequent than RobinHood.

Figure 11 shows the P99 request latency under increasing load on backend 1, where high latency is *correlated with query rate*, which is not typical in production systems. Figure 11(a) shows that, when load increases slowly, RobinHood never violates a 150ms SLO. OneRFpolicy and TAO_{++} lead to lower P99s than when latency is uncorrelated, but still violate the 150ms SLO more than 28% of the time. Of the research systems, $Cliffhgr_{++}$ is the best with about 8.2% SLO violations.

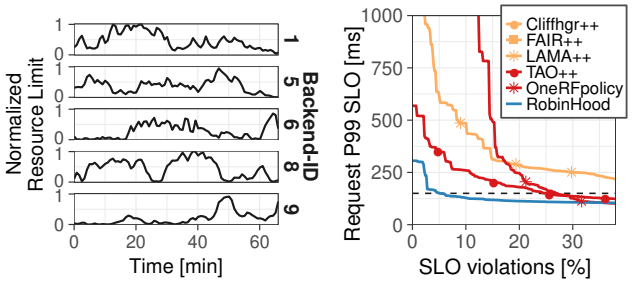
Figure 11(b) shows that, when load increases quickly, RobinHood never violates the SLO. The second best system, OneRFpolicy, violates the SLO more than 33% of the time.

5.4 How robust is RobinHood to simultaneous latency spikes?

To test the robustness of RobinHood, we allow the backend resource limits to vary randomly and measure RobinHood's ability to handle a wide range of latency imbalance patterns.

Experimental setup. We adjust resource limits over time for the same backends and over the same ranges as those used in the experiments from Section 5.2. However, rather than inducing latency spikes similar to those in the OneRF production system, we now allow resource limits to vary randomly. Hence, each backend will have multiple periods of high and low latency over the course of the experiment. Additionally, multiple backends may experience high latency at the same time. To generate this effect, we randomly increase or decrease each backend's resource limit with 50% probability every 20 seconds.

Experimental results. Figure 12 shows the results of our robustness experiments. Figure 12(a) shows the backend resource limits (normalized to the limits in Section 5.2) over time for each of the backends that were resource limited during the experiment. Note that at several times during the experiment, *multiple backends* were highly limited at the same time, making it more difficult to maintain low request tail latency.



(a) Randomized Resource Limits. (b) SLO violations.

Figure 12: Results from robustness experiments in which backend resource limits vary randomly, see (a). In this challenging scenario, RobinHood still meets 150ms P99 SLO 95% of the time, see (b).

Figure 12(b) shows the rate of SLO violations for each caching system during the robustness experiments. In this challenging scenario, RobinHood violates a 150ms SLO only 5% of the time. The next best policy, TAO_{++}, violates the SLO more than 24% of the time. RobinHood also helps during parts of the experiment where all backends are severely resource constrained. Overall, RobinHood's maximum P99 latency does not exceed 306ms whereas the next best policy, TAO_{++}, exceeds 610ms.

We observe that the there is no single "second best" caching system: the order of the competitors OneRFpolicy, TAO_{++}, and $LAMA_{++}$ is flipped between Figures 12 and 8. In Figure 12(b), TAO_{++} performs well by coincidence and not due to an inherent advantage of static allocations. TAO_{++}'s static allocation is optimized for the first part of the experiment shown in Figure 7, where a latency spike occurs on backend 6. Coincidentally, throughout our randomized experiment, backend 6 is also severely resource limited, which significantly boosts TAO_{++}'s performance.

5.5 How much space does RobinHood save?

Figure 13 shows RobinHood's allocation per backend in the experiments from Sections 5.2 and 5.4. To get an

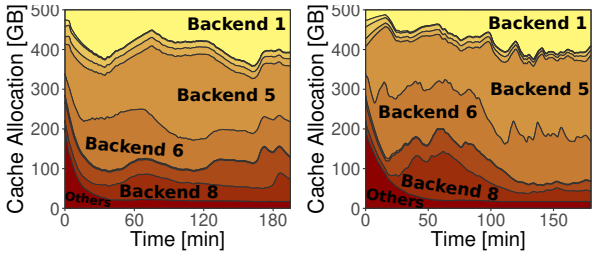


Figure 13: RobinHood's overall cache allocation during the experiments from Sections 5.2 and 5.4.

estimate of how much space RobinHood saves over other caching systems, we consider what static allocation would be required by TAO_{++} in order to provide each backend with its maximum allocation under RobinHood. This significantly underestimates RobinHood's advantage, since it assumes the existence of an oracle that knows RobinHood's allocations ahead of time. Even given advance knowledge of these allocations, TAO_{++} would need 73% more cache space than RobinHood.

5.6 What is the overhead of running RobinHood?

We consider three potential sources of overhead.

Network overhead. In our implementation of RobinHood, application servers send request statistics to an RBC server (Section 3) once every second. These updates include the request latency (32-bit integer) and the ID of the request's blocking backend (32-bit integer). Given a request rate of 1000 requests per second per application server, this amounts to less than 8 KB/s.

CPU and memory overhead. RobinHood adds a lightweight controller to each application server (Section 3). Throughout all experiments, the controller's CPU utilization overhead was too small to be measured. The memory overhead including RobinHood's controller is less than 25 KB. However, we measured bursts of memory overhead up to several MBs. This is due to memcached not freeing pages immediately after completing deallocation requests. Future implementations could address these bursts by refining the memcached resizing mechanism.

Query hit latency overhead. In multi-threaded caching systems, such as memcached, downsizing a partition will cause some concurrent cache operations to block (Section 4). We quantify this overhead by measuring the P99 cache hit latency for queries in RobinHood and OneRF for the five backends with the largest change in partition sizes (backends 1, 5, 6, 8, and 9). These measurements are shown in Figure 14. RobinHood increases the P99 cache hit latency for queries by 13% to 28% for the five backends, but does not significantly affect the other backends. Importantly, recall that request latency is different from query latency. The cause of high *request* tail latency is almost solely due to cache misses. Consequently, these hit latencies do not increase the request latency of RobinHood even for low percentiles (cf. the P50 request latency in Figure 8c).

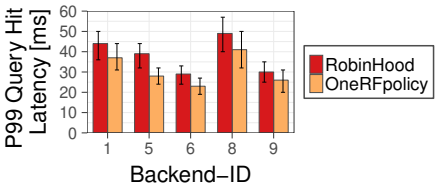


Figure 14: The cache hit latencies under RobinHood and the OneRFpolicy. The small overhead introduced by RobinHood does not affect request tail latency.

Query miss latency overhead. RobinHood can increase the load on cache-rich backends. Across all experiments, the worse-case increase of an individual backend (backend 20, the least queried backend), is 2.55× over OneRF. Among the top 5 backends, RobinHood never increases query latencies by more than 61%. On the other hand, RobinHood improves the P99 query latency by more than 4× for the overloaded backend during the first latency spike in Figure 7. By sacrificing the performance of cache-rich backends, RobinHood frees up cache space to allocate towards cache-poor backends that are contributing to slow request latency. This trade-off significantly improves the request latency both at the P99 as well as at other percentiles (Section 5.2).

6 Discussion

We have seen that RobinHood is capable of meeting a 150ms SLO for the OneRF workload even under challenging conditions where backends simultaneously become overloaded. Many other systems, e.g., at Facebook [20], Google [26], Amazon [27], and Wikipedia [14], use a similar multitier architecture where a request depends on many queries. However, these other systems may have different optimization goals, more complex workloads, or slight variations in system architecture compared to OneRF. In this section, we discuss some of the challenges that may arise when incorporating RobinHood into these other systems.

Non-convex miss curves. Prior work has observed non-convex miss ratio curves (a.k.a. performance cliffs) for some workloads [12, 25, 73, 80]. This topic was also frequently raised in our discussions with other companies. While miss ratio curves in our experiments are largely convex, RobinHood does not fundamentally rely on convexity. Specifically, RobinHood never gets stuck, because it ignores the miss ratio slope. Nevertheless, non-convexities can lead to inefficiency in RobinHood's allocation. If miss ratio curves are highly irregular (step functions), we suggest convexifying miss ratios using existing techniques such as Talus [12] and Cliffhanger [25].

Scaling RobinHood to more backend systems and higher request rates. The RobinHood algorithm scales

linearly in the number of backend systems and thus can support hundreds of backends (e.g. services in a microservice architecture). Even at high request rates, RobinHood's overhead is only a few MB/s for up to a million requests per second (independent of the query rate). At a sufficiently high request rate, RobinHood's central RBC server may become the bottleneck. However, at this scale, we expect that it will no longer be necessary to account for every request when calculating the RBC. Sampling some subset of the traffic will still produce a P99 estimate with enough observations to accurately depict the system state. It is also worth noting that typical production systems already have latency measurement systems in place [77] and thus would not require a dedicated RBC server.

Interdependent backends. RobinHood assumes that query latencies are independent across different backends. In some architectures, however, multiple backends share the same underlying storage system [3]. If this shared storage system were the bottleneck, allocating cache space to just one of the backends may be ineffective. RobinHood needs to be aware of such interdependent backends. A future version of RobinHood could fix this problem by grouping interdependent backends into a single unit for cache allocations.

Multiple webservices with shared backends. Optimizing tail latencies across multiple webservices which make use of the same, shared backend systems is challenging. RobinHood can introduce additional challenges. For example, one webservice running RobinHood may increase the load significantly on a shared backend which can negatively affect request latencies in a second webservice. This could arise if the two services see differently structured requests—the shared backend could seem unimportant to one webservice but be critical to another. If both webservices run RobinHood, a shared backend's load might oscillate between low and high as the two RobinHood instances amplify the effect of each other's allocation decisions. A solution to this problem could be to give RobinHood controllers access to the RBC servers of both webservices (effectively running a global RobinHood instance). If this is impossible, additional constraints on RobinHood's allocation decisions could be necessary. For example, we can constrain RobinHood to assign at least as much capacity to the shared backend as it would get in a system without RobinHood.

Distributed caching. Many large webservices rely on a distributed caching layer. While these layers have access to large amounts of cache capacity, working sets typically still do not fit into the cache and the problem of tuning partition sizes remains [20]. RobinHood can accommodate this scenario with solely a configuration change, associating a RobinHood controller with each cache rather than each application server. We have tested RobinHood in this configuration and verified the feasibility of our proposal. However, distributed caching leads to the known problem of cache hotspots under the OneRF workload, regardless of whether or not RobinHood is running (see Section 4.2 and [20, 77]). Addressing this issue is outside the scope of this work, and hence we focus on the cache topology used by OneRF rather than a distributed caching layer.

Scenarios where cache repartitioning is not effective. If the caching layer is severely underprovisioned, or if the workload is highly uncacheable, repartitioning the cache might not be sufficient to reduce P99 request latency. However, we note that RobinHood's key idea—allocating resources to backends which affect P99 request latency—can still be exploited. For instance, if caching is ineffective but backends can be scaled quickly, the RBC metric could be used to drive these scaling decisions in order to reduce request tail latency. Even if backends are not scalable, RBC measurements collected over the course of a day could inform long-term provisioning decisions.

Performance goals beyond the P99. Depending on the nature of the application, system designers may be concerned that a using single optimization metric (e.g., P99) could lead to worse performance with respect to other metrics (e.g., the average request latency). However, RobinHood explicitly optimizes whatever optimization metric is used to calculate the RBC. Hence, it is possible to use other percentiles or even multiple percentiles to calculate the RBC by choosing the set S accordingly (see Section 3 for a definition of S). Conceptually, RobinHood is modular with regard to both the resources it allocates and with regard to the metric that is used to drive these allocations.

7 Related Work

A widely held opinion is that "caching layers ... do not directly address tail latency, aside from configurations where it is guaranteed that the entire working set of an application can reside in a cache" [26]. RobinHood is the first work that shows that *caches can directly address tail latency even if working sets are much larger than the cache size*. Thus, RobinHood stands at the intersection of two bodies of work: caching and tail latency reduction.

Caching related work. Caching is a heavily studied area of research ranging from theory to practice [10]. For the most part, the caching literature has primarily focused on improving hit ratios (e.g., [2, 8, 13, 15–17, 21, 22, 42, 57, 87]). Prior work has also investigated strategies for dynamically partitioning a cache to maximize overall hit ratio (e.g., [1, 24, 25, 40, 60, 75]) or to provide a weighted or fair hit ratio to multiple workloads (e.g., [19,46,65,85]). Importantly, while hit ratio is a good proxy for average latency, it does not capture the effect of tail latency, which is dominated by the backend system performance.

Another group of caching policies incorporates "miss cost" (such as retrieval latency) into eviction decisions [18, 21, 33, 52, 64, 70, 81, 86]. As discussed in Section 2.2.2,

the OneRF workload does not meet the premise of cost-aware caching. Specifically, all these systems assume that the retrieval latency is correlated (fixed) per object. At OneRF, latency is highly variable over time and not correlated with specific objects.

The most relevant systems are LAMA [40] and Hyperbolic [18]. LAMA partitions the cache by backend and seeks to balance the average latency across backends. Hyperbolic does not support partitions, but allows estimating a metric across a group of related queries such as all queries going to the same backend. This enables Hyperbolic to work similarly to LAMA. Both LAMA and Hyperbolic are represented optimistically by LAMA_{++} in our evaluation. Unfortunately, LAMA_{++} leads to high P99 request latency because the latency of individual queries is typically not a good indicator for the overall request tail latency (see Section 2). Unlike LAMA or Hyperbolic, RobinHood directly incorporates the request structure in its caching decisions.

Another branch of works seeks to improve the caching system itself, e.g., the throughput [31, 66], the latency of cache hits [67], cache-internal load balancing [32, 43], and cache architecture [31, 55, 72]. However, these works are primarily concerned with the performance of cache hits rather than cache misses, which dictate the overall request tail latency.

Tail latency related work. Reducing tail latency and mitigating stragglers is an important research area that has received much attention in the past decade. Existing techniques can be subdivided into the following categories: redundant requests, scheduling techniques, auto-scaling and capacity provisioning techniques, and approximate computing. Our work serves to introduce a fifth category: *using the cache to reduce tail latency.*

A common approach to mitigating straggler effects is to send redundant requests and use the first completed request [5–7,45,69,78,79,82,84,88]. When requests cannot be replicated, prior work has proposed several scheduling techniques, e.g., prioritization strategies [36, 89, 92], load balancing techniques [48, 53, 83], and systems that manage queueing effects [4, 28, 29, 54, 62, 68, 74]. These are useful techniques for cutting long tail latencies, but fundamentally, they still have to send requests to backend systems, whereas our new caching approach eliminates a fraction of traffic to backend systems entirely.

While these first two approaches consider systems with static resource constraints, other works have considered adjusting the overall compute capacity to improve tail latency. These techniques include managing the compute capacity through auto-scaling and capacity provisioning for clusters [34,45,47,58,90,91], and adjusting the power and/or compute (e.g., number of cores) allocated to performing the computation [37, 38, 49, 76]. Alternatively, there is a branch of tail latency reduction work known as approximate computing, which considers reducing the computational requirements by utilizing lower quality results [6, 23, 45, 51]. Importantly, these are all orthogonal approaches for reducing tail latency, and our work is proposing a new type of technique that can be layered on top of these existing techniques.

Why RobinHood is different. RobinHood is unique in several ways. First, it is the only system to utilize the cache for reducing overall request tail latency. Second, RobinHood is the only caching system that takes request structure into account. Third, by operating at the caching layer, RobinHood is uniquely situated to influence many diverse backend systems without requiring any modifications to the backend systems.

8 Conclusion

This paper addresses two problems facing web service providers who seek to maintain low request tail latency. The first problem is to determine the best allocation of resources in multitier systems which serve structured requests. To deal with structured requests, RobinHood introduces the concept of the request blocking count (RBC) for each backend, identifying which backends require additional resources. The second problem is to address latency imbalance across stateful backend systems which cannot be scaled directly to make use of the additional resources. RobinHood leverages the existing caching layer present in multitiered systems, differentially allocating cache space to the various backends in lieu of being able to scale them directly.

Our evaluation shows that RobinHood can reduce SLO violations from 30% to 0.3% for highly variable workloads such as OneRF. RobinHood is also lightweight, scalable, and can be deployed on top of an off-the-shelf software stack. The RobinHood caching system demonstrates how to effectively identify the root cause of P99 request latency in the presence of structured requests. Furthermore, RobinHood shows that, contrary to popular belief, a properly designed caching layer *can* be used to reduce higher percentiles of request latency.

Acknowledgments

We thank Jen Guriel, Bhavesh Thaker, Omprakash Maity, and everyone on the OneRF team at Microsoft. We also thank the anonymous reviewers, and our shepherd, Frans Kaashoek, for their feedback. This paper was supported by NSF-CSR-180341, NSF-XPS-1629444, NSF-CMMI-1538204, and a Faculty Award from Microsoft.

References

[1] C. L. Abad, A. G. Abad, and L. E. Lucio. Dynamic memory partitioning for cloud caches with heteroge-

neous backends. In *ACM ICPE*, pages 87–90, New York, NY, USA, 2017. ACM.

[2] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM*, pages 293–305, 1996.

[3] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX ATC*, pages 91–102, 2013.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 19–19, 2012.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[6] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming stragglers in approximation analytics. In *USENIX NSDI*, pages 289–302, Seattle, WA, 2014. USENIX Association.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, pages 265–278, Berkeley, CA, USA, 2010. USENIX Association.

[8] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, 2000.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, pages 53–64, 2012.

[10] A. Balamash and M. Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004.

[11] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX NSDI*, pages 389–403, 2018.

[12] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *IEEE HPCA*, pages 64–75, 2015.

[13] D. S. Berger, N. Beckmann, and M. Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *POMACS*, 2(2):32, 2018.

[14] D. S. Berger, B. Berg, T. Zhu, and M. Harchol-Balter. The case of dynamic cache partitioning for tail latency, March 2017. Poster presented at USENIX NSDI.

[15] D. S. Berger, P. Gland, S. Singla, and F. Ciucu. Exact analysis of TTL cache networks. *Perform. Eval.*, 79:2 – 23, 2014. Special Issue: Performance 2014.

[16] D. S. Berger, S. Henningsen, F. Ciucu, and J. B. Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Perform. Eval. Rev.*, 43(2):57–59, Sept. 2015.

[17] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, pages 483–498, Berkeley, CA, USA, 2017. USENIX Association.

[18] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[19] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *IEEE ICPP*, pages 749–758, 2015.

[20] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.

[21] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[22] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking*, pages 114–123, 2001.

[23] M. Chow, K. Veeraraghavan, M. Cafarella, and J. Flinn. Dqbarge: Improving data-quality tradeoffs in large-scale internet services. In *USENIX OSDI*, pages 771–786, Savannah, GA, 2016. USENIX Association.

[24] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: dynamic cloud caching. In *USENIX HotCloud*, 2015.

[25] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, 2016.

[26] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, volume 41, pages 205–220, 2007.

[28] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *ACM SoCC*, pages 497–509, New York, NY, USA, 2016. ACM.

[29] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX ATC*, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association.

[30] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ACM ASPLOS*, pages 127–144, 2014.

[31] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[32] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[33] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *USENIX FAST*, pages 5–5, Berkeley, CA, USA, 2002. USENIX Association.

[34] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM TOCS*, 30(4):14, 2012.

[35] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[36] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *USENIX NSDI*, pages 9–21, 2015.

[37] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, pages 161–175, New York, NY, USA, 2015. ACM.

[38] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *IEEE/ACM MICRO*, pages 625–638, New York, NY, USA, 2017. ACM.

[39] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[40] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC*, pages 57–69, 2015.

[41] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[42] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[43] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *Proceedings of the International Conference on Autonomic Computing*, pages 33–43, San Jose, CA, 2013. USENIX.

[44] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, 2013.

[45] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, New York, NY, USA, 2013. ACM.

[46] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer. Adaptive insertion policies for managing shared caches. In *ACM PACT*, pages 208–219, 2008.

[47] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 435–448, New York, NY, USA, 2015. ACM.

[48] S. A. Javadi and A. Gandhi. DIAL: reducing tail latencies for cloud applications via dynamic

interference-aware load balancing. In *International Conference on Autonomic Computing*, pages 135–144, 2017.

[49] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *IEEE/ACM MICRO*, pages 598–610, Dec 2015.

[50] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM*, 41(1):102–108, 2011.

[51] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica. Hold 'em or fold 'em?: Aggregation queries under performance variations. In *ACM EUROSYS*, pages 7:1–7:14, 2016.

[52] C. Li and A. L. Cox. GD-Wheel: A cost-aware replacement policy for key-value stores. In *ACM EUROSYS*, pages 5:1–5:15, 2015.

[53] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *ACM PPoPP*, pages 14:1–14:13, New York, NY, USA, 2016.

[54] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM SoCC*, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[55] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, pages 429–444, 2014.

[56] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM ISCA*, volume 42, pages 301–312, 2014.

[57] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[58] A. H. Mahmud, Y. He, and S. Ren. Bats: Budget-constrained autoscaling for cloud performance optimization. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 232–241, Oct 2015.

[59] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ACM ISCA*, volume 39, pages 319–330, 2011.

[60] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.

[61] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[62] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *ACM SOSP*, pages 69–84, New York, NY, USA, 2013. ACM.

[63] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, pages 307–318, 2014.

[64] R. Prabhakar, S. Srikantaiah, C. Patrick, and M. Kandemir. Dynamic storage cache allocation in multi-server architectures. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[65] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *USENIX NSDI*, pages 393–406, 2016.

[66] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM MICRO*, pages 423–432, 2006.

[67] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[68] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *EuroSys*, pages 95–110, New York, NY, USA, 2017. ACM.

[69] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *ACM SIGCOMM*, pages 379–392, New York, NY, USA, 2015. ACM.

[70] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM TON*, 8:158–170, 2000.

[71] E. Rocca. Running Wikipedia.org, June 2016. available `https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf` accessed 09/12/16.

[72] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *ACM SoCC*, pages 174–181, New York, NY, USA, 2015. ACM.

[73] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[74] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, pages 513–527, Oakland, CA, 2015. USENIX Association.

[75] J. Tan, G. Quan, K. Ji, and N. Shroff. On resource pooling and separation for lru caching. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1):5:1–5:31, Apr. 2018.

[76] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *ACM/IEEE MICRO*, pages 585–597, New York, NY, USA, 2015. ACM.

[77] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *USENIX OSDI*, pages 635–650, 2016.

[78] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *ACM CoNEXT*, pages 283–294, New York, NY, USA, 2013. ACM.

[79] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *ACM HotNets*, pages 13–18, New York, NY, USA, 2012. ACM.

[80] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, pages 487–498, 2017.

[81] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977–986, 1997.

[82] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *USENIX NSDI*, pages 543–557, Oakland, CA, 2015. USENIX Association.

[83] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *USENIX NSDI*, pages 329–341, Lombard, IL, 2013. USENIX.

[84] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *ACM SoCC*, pages 26:1–26:14, New York, NY, USA, 2014. ACM.

[85] C. Ye, J. Brock, C. Ding, and H. Jin. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *International Journal of Parallel Programming*, 45(1):30–44, 2017.

[86] N. E. Young. On-line file caching. In *ACM SODA*, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[87] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. LRC: dependency-aware cache management for data analytics clusters. *CoRR*, abs/1703.08280, 2017.

[88] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[89] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *SIGPLAN Not.*, 51(4):33–47, Mar. 2016.

[90] T. Zhu, D. S. Berger, and M. Harchol-Balter. SNC-Meister: Admitting more tenants with tail latency SLOs. In *ACM SoCC*, pages 374–387, 2016.

[91] T. Zhu, M. A. Kozuch, and M. Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providing tail latency slo guarantees. In *ACM SoCC*, pages 598–610, New York, NY, USA, 2017. ACM.

[92] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *ACM SoCC*, pages 29:1–29:14, New York, NY, USA, 2014. ACM.

Noria: dynamic, partially-stateful data-flow for high-performance web applications

Jon Gjengset * Malte Schwarzkopf * Jonathan Behrens Lara Timbó Araújo
Martin Ek† Eddie Kohler‡ M. Frans Kaashoek Robert Morris
MIT CSAIL † *Norwegian University of Science and Technology* ‡ *Harvard University*

Abstract

We introduce *partially-stateful data-flow*, a new streaming data-flow model that supports eviction and reconstruction of data-flow state on demand. By avoiding state explosion and supporting live changes to the data-flow graph, this model makes data-flow viable for building long-lived, low-latency applications, such as web applications. Our implementation, *Noria*, simplifies the backend infrastructure for read-heavy web applications while improving their performance.

A Noria application supplies a relational schema and a set of parameterized queries, which Noria compiles into a data-flow program that pre-computes results for reads and incrementally applies writes. Noria makes it easy to write high-performance applications without manual performance tuning or complex-to-maintain caching layers. Partial statefulness helps Noria limit its in-memory state without prior data-flow systems' restriction to windowed state, and helps Noria adapt its data-flow to schema and query changes while on-line. Unlike prior data-flow systems, Noria also shares state and computation across related queries, eliminating duplicate work.

On a real web application's queries, our prototype scales to 5× higher load than a hand-optimized MySQL baseline. Noria also outperforms a typical MySQL/memcached stack and the materialized views of a commercial database. It scales to tens of millions of reads and millions of writes per second over multiple servers, outperforming a state-of-the-art streaming data-flow system.

1 Introduction

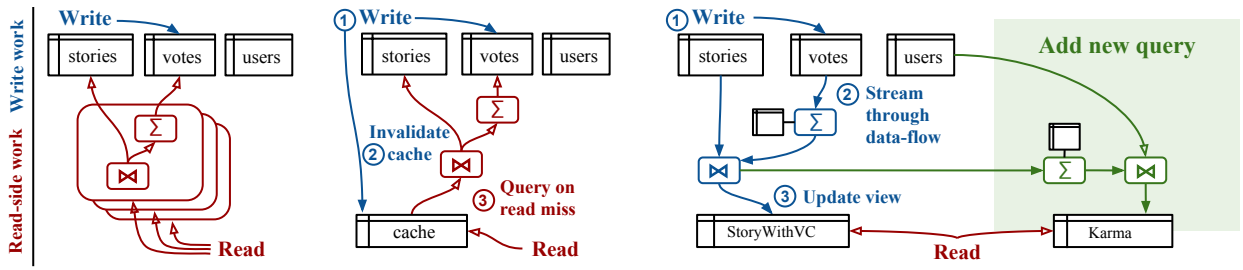
Web applications must serve many users at low latency. They respond to each user request using data queried from backend stores, usually relational databases. The vast majority of such store accesses are reads, and evaluating them as repeated queries over the normalized schema of a relational database is inefficient [54, 57]. Hence, many applications explicitly include pre-computed query results in their database schemas, or cache such results in separate key-value stores [8, 54]. For example, the Lobsters news aggregator [43] stores stories' computed vote counts and "hotness" in separate table columns to avoid re-computing them on every page load [42]. As each vote is reflected in several places, application logic must explicitly update computed columns every time a value changes. Hence, pre-computation complicates both application reads and writes. In general, developers must choose between convenient, but slow, "natural" relational queries (*e.g.*, with inline aggregations), and increased performance at the cost of application and deployment complexity (*e.g.*, due to caching).

Noria applications do not need to choose. Noria exposes a high-level query interface (SQL), but unlike in conventional systems, Noria accelerates the execution of even complex natural queries by answering with pre-computed results where possible. At its core, Noria runs a continuous, but dynamically changing, *data-flow computation* that combines the persistent store, the cache, and elements of application logic. Each write to Noria streams through a joint data-flow graph for the current queries and incrementally updates the cached, eventually-consistent internal state and query results.

Making this approach work for web applications is challenging. A naïve implementation might maintain unbounded pre-computed state, causing unacceptable space and time overhead, so Noria must limit its state size. Writes can update many pre-computed results, so Noria must ensure that writes are fast and avoid unnecessary work. Finally, since many web applications frequently change their queries [20, 61], Noria must accommodate changes without iterating over all data.

Existing data-flow systems either cannot perform fine-grained incremental updates to state [36, 52, 75], or limit the growth of operator state using "windowed" state (*e.g.*, this week's stories). This bounds their memory footprint but prohibits reading older data [11, 39, 46, 51]. Noria's data-flow operator state is *partial* instead of windowed, retaining only the subset of records that the application has queried. This is possible thanks to a new, *partially-stateful* data-flow model: when in need of missing state, operators request an *upquery* that derives the missing records from upstream state. Ensuring correctness with this model requires careful attention to invariants, as ordinary updates and upqueries can race. With-

* equal contribution



(a) Classic database operation with compute on reads.

(b) Two-tier stack with demand-filled cache [54, §2].

(c) Noria: stateful data-flow operators pre-compute data for reads incrementally; data-flow change supports new queries.

Figure 1: Overview of how current website backends and Noria process frontend reads and writes.

out care, such races could produce permanently incorrect state, and therefore incorrect cached query results.

The state that Noria keeps is similar to a materialized view, and its data-flow processing is akin to view maintenance [2, 37]. Noria demonstrates that, contrary to conventional wisdom, maintaining materialized views for all application queries is feasible. This is possible because partially-stateful operators can evict rarely-used state, and discard writes for that state, which reduces state size and write load. Noria further avoids redundant computation and state by jointly optimizing its queries to merge overlapping data-flow subgraphs.

Few existing streaming data-flow systems can change their queries and input schemas without downtime. For example, Naiad must re-start to accommodate changes, and Spark's Structured Streaming must restart from a checkpoint [18]. Noria, by contrast, adapts its data-flow to new queries without interrupting existing clients. It applies changes while retaining existing state and while remaining live for reads throughout. Writes from current clients see sub-second interruptions in the common case.

Noria's techniques remain compatible with traditional parallel and distributed data-flow, and allow Noria to parallelize and scale fine-grained, partially materialized view maintenance over multiple cores and machines.

In summary, Noria makes four principal contributions:

1. the partially-stateful data-flow model, its correctness invariants, and a conforming system design;
2. automatic merge-and-reuse techniques for data-flow subgraphs in joint data-flows over many queries, which reduce processing cost and state size;
3. near-instantaneous, dynamic transitions for data-flow graphs in response to changes to queries or schema without loss of existing state; and
4. a prototype implementation and an evaluation that demonstrates that practical web applications benefit from Noria's approach.

Our Noria prototype exposes a backwards-compatible MySQL protocol interface and can serve real web applications with minimal changes, although its benefits increase for Noria-optimized applications. When serving the Lobsters web application on a single Amazon EC2 VM, our prototype outperforms the default MySQL-based backend by 5× while simultaneously simplifying the application (§8.1). For a representative query, our prototype outperforms the widely-used MySQL/memcached stack and the materialized views of a commercial database by 2–10× (§8.2). It also scales the query to millions of writes and tens of millions of reads per second on a cluster of EC2 VMs, outperforming a state-of-the-art data-flow system, differential dataflow [46, 51] (§8.3). Finally, our prototype adapts the data-flow without any perceptible downtime for reads or writes when transitioning the same query to a modified version (§8.5).

Nevertheless, our current prototype has some limitations. It only guarantees eventual consistency; its eviction from partial state is randomized; it is inefficient for sharded queries that require shuffles in the data-flow; and it lacks support for some SQL keywords. We plan to address these limitations in future work.

2 Background

We now explain how current website backends and Noria process data. Figure 1 shows an overview.

Many web applications use a **relational database** to store and query data (Figure 1a). Page views generate database queries that frequently require complex computation, and the query load tends to be read-heavy. Across one month of traffic data from a HotCRP site and the production deployment of Lobsters [32], 88% to 97% of queries are reads (`SELECT` queries), and these reads consume 88% of total query execution time in HotCRP. Since read performance is important, application developers often manually optimize it. For example, Lobsters stores individual votes for stories in a `votes` table, but also stores per-story vote counts as a column in the `stories` table. This speeds up read queries of vote counts, but "de-normalizes" the schema and complicates vote writes, which must update the derived counts.

Websites often deploy an **in-memory key-value cache** (like Redis, memcached, or TAO [8]) to speed up common-case read queries (Figure 1b). Such a cache avoids re-evaluating the query when the underlying records are unchanged. However, the application must invalidate or replace cache entries as the records change. This process is error-prone and requires complex application-side logic [37, 48, 57, 64]. For example, developers must carefully avoid performance collapse due to "thundering herds" (*viz.*, many database queries issued just after an invalidation) [54, 57]. Since the cache can return stale records, reads are eventually-consistent.

Some sites use **stream-processing systems** [13, 39] to maintain results for queries whose re-execution over all past data is infeasible. One major problem for these systems is that they must maintain *state* at some operators, such as aggregations. To avoid unbounded growth, existing systems "window" this state by limiting it to the most recent records. This makes it difficult for a stream processor to serve the general queries needed for websites, which need to access older as well as recent state. Moreover, stream processors are less flexible than a database that can execute any relational query on its schema: introducing a new query often requires a restart.

Noria, as shown in Figure 1c, combines the best of these worlds. It supports the fast reads of key-value caches, the efficient updates and parallelism of streaming data-flow, and, like a classic database, supports changing queries and base table schemas without downtime.

3 Noria design

Noria is a stateful, dynamic, parallel, and distributed data-flow system designed for the storage, query processing, and caching needs of typical web applications.

3.1 Target applications and deployment

Noria targets read-heavy applications that tolerate eventual consistency. Many web applications fit this model: they accept the eventual consistency imposed by caches that make common-case reads fast [15, 19, 54, 72]. Noria's current design primarily targets relational operators, rather than the iterative or graph computations that are the focus of other data-flow systems [46, 51], and processes structured records in tabular form [12, 16]. Large blobs (*e.g.*, videos, PDF files) are best stored in external blob stores [7, 24, 50] and referenced by Noria's records.

Noria runs on one or more multicore servers that communicate with clients and with one another using RPCs. A Noria deployment stores both *base tables* and *derived views*. Roughly, base tables contain the data typically stored persistently, and derived views hold data an application might choose to cache. Compared to conventional database use, Noria base tables might be smaller, as Noria derives data that an application may otherwise precompute and store in base tables for performance. Views, by contrast, will likely be larger than a typical cache footprint, because Noria derives more data, including some intermediate results. Noria stores base tables persistently on disk, either on one server or sharded across multiple servers, but stores views in server memory. The application's working set in these views should fit in memory for good performance, but Noria reduces memory use by only materializing records that are actually read, and by evicting infrequently-accessed data.

```
/* base tables */
CREATE TABLE stories
  (id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);
/* internal view: vote count per story */
CREATE INTERNAL VIEW VoteCount AS
  SELECT story_id, COUNT(*) AS vcount
    FROM votes GROUP BY story_id;
/* external view: story details */
CREATE VIEW StoriesWithVC AS
  SELECT id, author, title, url, vcount
    FROM stories
    JOIN VoteCount ON VoteCount.story_id = stories.id
   WHERE stories.id = ?;
```

Figure 2: Noria program for a key subset of the Lobsters news aggregator [43] that counts users' votes for stories.

3.2 Programming interface

Applications interact with Noria via an interface that resembles parameterized SQL queries. The application supplies a *Noria program*, which registers base tables and views with parameters supplied by the application when it retrieves data. Figure 2 shows an example Noria program for a Lobsters-like news aggregator application (? is a parameter). The Noria program includes base table definitions, *internal* views used as shorthands in other expressions, and *external* views that the application later queries. Internally, Noria instantiates a data-flow to continuously process the application's writes through this program, which in turn maintains the external views.

To retrieve data, the application supplies Noria with an external view identifier (*e.g.*, `StoriesWithVC`) and one or more sets of parameter values. Noria then responds with the records in the view that match those values. To modify records in base tables, the application performs insertions, updates, and deletions, similar to a SQL database. Noria applies these changes to the appropriate base tables and updates dependent views.

The application may change its Noria program to add new views, to modify or remove existing views, and to adapt base table schemas. Noria expects such changes to be common and aims to complete them quickly. This contrasts with most previous data-flow systems, which lack support for efficient changes without downtime.

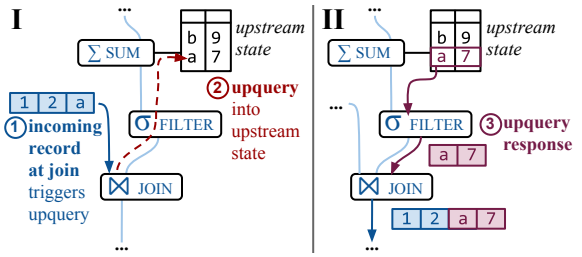


Figure 3: Noria's data-flow operators can query into upstream state: a join issues an upquery (**I**) to retrieve a record from upstream state to produce a join result (**II**).

In addition to its native SQL-based query interface, Noria provides an implementation of the MySQL binary protocol, which allows existing applications that use prepared statements against a MySQL database to interact with Noria without further changes. The adapter turns ad-hoc queries and prepared SQL statements into writes to base tables, reads from external views, and incrementally effects Noria program changes. Noria supports much, but not all, SQL syntax. We discuss the experience of building and porting applications in §7.

3.3 Data-flow execution

Noria's data-flow is a directed acyclic graph of relational operators such as aggregations, joins, and filters. Base tables are the roots of this graph, and external views form the leaves. Noria extends the graph with new base tables, operators, and views as the application adds new queries.

When an application write arrives, Noria applies it to a durable base table and injects it into the data-flow as an *update*. Operators process the update and emit derived updates to their children; eventually updates reach and modify the external views. Updates are *deltas* [46, 60] that can add to, modify, and remove from downstream state. For example, a count operator emits deltas that indicate how the count for a key has changed; a join may emit an update that installs new rows in downstream state; and a deletion from a base table generates a "negative" update that revokes derived records. Negative updates remove entries when Noria applies them to state, and retain their negative "sign" when combined with other records (*e.g.*, through joins). Negative updates hold exactly the same values as the positives they revoke and thus follow the same data-flow paths.

Noria supports *stateless* and *stateful* operators. Stateless operators, such as filters and projections, need no context to process updates; stateful operators, such as count, min/max, and top-k, maintain state to avoid inefficient re-computation of aggregate values from scratch. Stateful operators, like external views, keep one or more indexes to speed up operation. Noria adds indexes based on *indexing obligations* imposed by operator semantics; for example, an operator that aggregates votes by user ID requires a user ID index to process new votes efficiently.

In most stream processors, join operators keep a windowed cache of their inputs [3, 76], allowing an update arriving at one input to join with all relevant state from the other. In Noria, joins instead perform *upqueries*, which are requests for matching records from stateful ancestors (Figure 3): when an update arrives at one join input, the join looks up the relevant state by querying its other inputs. This reduces Noria's space overhead, since joins often need not store duplicate state, but requires care in the presence of concurrent updates, an issue further discussed in §4. Upqueries also impose indexing obligations that Noria detects and satisfies.

3.4 Consistency semantics

To achieve high parallel processing performance, Noria's data-flow avoids global progress tracking or coordination. An update injected by a base table takes time to propagate through the data-flow, and the update may appear in different views at different times. Noria operators and the contents of its external views are *eventually-consistent*. Eventual consistency is attractive for performance and scalability, and is sufficient for many web applications [15, 54, 72].

Noria does ensure that if writes quiesce, all external views eventually hold results that are the same as if the queries had been executed directly against the base table data. Making this work correctly requires some care. Like most data-flow systems, Noria requires that operators are deterministic functions over their own state and the inputs from their ancestors. In addition, Noria must avoid races between updates and upqueries; avoid reordering updates on the same data-flow path; and resolve races between related updates that arrive independently at multi-ancestor operators via different data-flow paths. Consider an `OR` that combines filters using a union operator, or a join between data-flow paths connected to the same base table: such operators' final output (and state) must be commutative over the order in which updates arrive at their inputs. The standard relational operators Noria supports have this property.

Web applications sometimes rely on database transactions, *e.g.*, to atomically update pre-computed values. Noria approach's is compatible with basic, optimistically-concurrent multi-statement transactions, but Noria also often obviates the need for them. For example, Lobsters uses transactions only to avoid write-write conflicts on vote counts and stories' "hotness" scores. A multi-statement transaction is required only because baseline Lobsters pre-computes hotness for performance. Noria instead computes hotness in the data-flow, which avoids write-write conflicts without a transaction, albeit at the cost of eventual consistency for reads. We

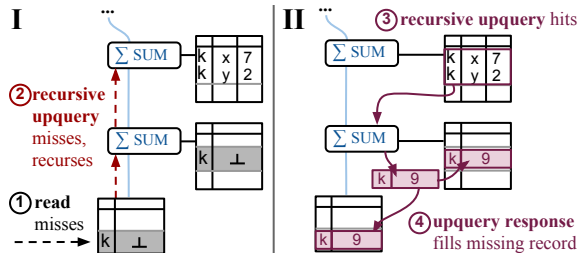


Figure 4: A partially-stateful view sends a *recursive upquery* to derive evicted state ($\perp$) for key k from upstream state (**I**); the response fills the missing state (**II**).

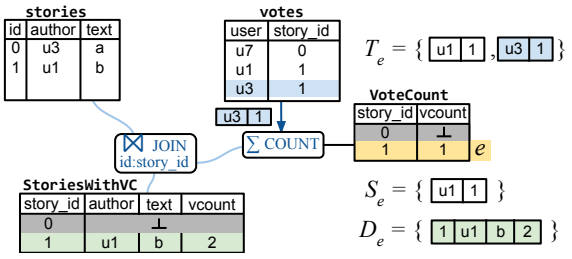


Figure 5: Definitions for partial state entry e (yellow) in `VoteCount`: an in-flight update from `votes` (blue) is in T_e, but not yet in S_e; the entry in `StoriesWithVC` is key-descendant from e via `story_id` (green).

omit further discussion of transactions with Noria in this paper; we plan to describe them in future work.

3.5 Challenges

An efficient Noria design faces two key challenges: first, it must limit the size of its state and views (§4); and second, changes to the Noria program must adapt the data-flow without downtime in serving clients (§5).

4 Partially-stateful data-flow

Noria must limit the size of its views, as the state for an application with many queries could exceed available memory and become too expensive to maintain.

The *partially-stateful data-flow model* lets operators maintain only a subset of their state. This concept of partial materialization is well-known for materialized views in databases [79, 80], but novel to data-flow systems. Partial state reduces memory use, allows eviction of rarely-used state, and relieves operators from maintaining state that is never read. Partially-stateful data-flow generalizes beyond Noria, but we highlight specific design choices that help Noria achieve its goals.

Partial state introduces new data-flow messages to Noria. *Eviction notices* flow forward along the update data-flow path; they indicate that some state entries will no longer be updated. Operators drop updates that would affect these evicted state entries without further processing or forwarding. When Noria needs to read from evicted state—for instance, when the application reads state evicted from an external view—Noria re-computes that state. This process sends *recursive upqueries* to the relevant ancestors in the graph (Figure 4). An ancestor that handles such an upquery computes the desired value (possibly after sending its own upqueries), then forwards a response that follows the data-flow path to the querying operator. When the upquery response eventually arrives, Noria uses it to populate the evicted entry. After the evicted entry has been filled, subsequent updates through the data-flow keep it up-to-date until it is evicted again.

For correctness, upqueries must produce eventually-consistent results. For performance, Noria should continue to process updates—including updates to the waiting operator—while (possibly slow) upqueries are in flight. These requirements complicate the design.

4.1 Data-flow model and invariants

We first describe high-level correctness invariants of Noria's partially-stateful data-flow. These invariants ensure that Noria remains eventually-consistent and never returns results contaminated by duplicate, missing, or spurious updates. Since Noria allows operators to execute in parallel to take advantage of multicore processors, these invariants must hold in the presence of concurrent updates and eviction notices. The invariants concern *state entries*, where a state entry models one record in one operator or view. Data-flow implementations derive state entry values from input records, possibly after multiple steps. For ease of expression, we model a state entry as the multiset of input records that produced that entry's value. Noria's eventual consistency requires that each state entry's contents approach the ideal set of input records that would produce the most up-to-date value.

Given some state entry e, we define:

- T_e is the set of all input records received so far that, in a correct implementation of the data-flow graph, would be used to compute e.
- S_e is either the multiset of input records *actually* used to compute in e, or $\perp$, which represents an evicted entry. We use a multiset so the model can represent potential bugs such as duplicate updates.
- D_e is the set of *key-descendant entries* of e. These are entries of operators downstream of e in the data-flow that depend on e through key lookup.

T_e and S_e are time-dependent, whereas the dependencies represented in D_e can be determined from the data-flow graph. If e is the `VoteCount` entry for some story in Figure 5, then T_e contains all input votes ever received for that story; S_e contains the updates represented in its `vcount`; and D_e includes its `StoriesWithVC` entry.

Correctness of partially-stateful data-flow relies on ensuring these invariants:

1. **Update completeness:** if $S_e \neq \bot$, then either all updates in $T_e - S_e$ are in flight toward e, or an eviction notice for e is in flight toward e.
2. **No spurious or duplicate updates:** $S_e \subseteq T_e$.
3. **Descendant eviction:** if $S_e = \bot$, then for all $d \in D_e$, either $S_d = \bot$, or an eviction notice for d is in flight toward d's operator.
4. **Eventual consistency:** if T_e stops growing, then eventually either $S_e = T_e$ or $S_e = \bot$.

We now explain the mechanisms that Noria uses to realize this data-flow model and maintain the invariants.

4.2 Update ordering

Noria uses update ordering to ensure eventual consistency without global data-flow coordination. Each operator totally orders all updates and upquery requests it receives for an entry; and, critically, the downstream data-flow ensures that all updates and upquery responses from that entry are processed by all consumers in that order. Thus, if the operator orders update u_1 before u_2, then every downstream consumer likewise processes updates derived from u_1 before those derived from u_2. Noria data-flows can split and merge (*e.g.*, at joins), but update ordering and operator commutativity ensure that the eventual result is correct independent of processing order.

4.3 Join upqueries

Join operators use upqueries (§3.3): when an update arrives at one input, the join upqueries its other input for the corresponding records, and combines them with the update. Join upqueries reach the next upstream stateful operator, which computes a snapshot of the requested state entry and forwards it along the data-flow to the querying join. Intermediate operators process the response as appropriate. Unlike normal updates, upquery responses follow the single path back to the querying operator without forking. Upquery responses also commute neither with each other nor with previous updates. This introduces a problem for join update processing, since *every* such update requires an upquery that produces non-commutative results, yet must produce an update that *does* commute.

Noria achieves this by ensuring that no updates are in flight between the upstream stateful operator and the join when a join upquery occurs. To do so, Noria limits the scope of each join upquery to an operator chain processed by a single thread. Noria executes updates on other operator chains in parallel with join upqueries.

This introduces a trade-off between parallelism and state duplication: join processing must stay within a single operator chain, so copies of upstream state may be required in each operator chain that contains a join.

4.4 Eviction and recursive upqueries

Evicted state introduces new challenges for Noria's data-flow. If the application requests evicted state, Noria must use recursive upqueries to fill it in. Moreover, operators now encounter evicted state when they handle updates. These factors influence the Noria design in several ways.

First and simplest, Noria operators drop updates that encounter evicted entries. This reduces the time spent processing updates downstream, but necessitates the descendant eviction invariant: operators downstream of an evicted entry never see updates for that entry, so they must evict their own dependent entries lest they remain permanently out of date.

Second, recursive upqueries now occasionally cascade up in the data-flow until they encounter the necessary state—in the worst case, up to base tables. Responses then flow forward to the querying operator. Upquery results are snapshots of operator state, and do not commute with updates. For unbranched chains, update ordering (§4.2) and the fact that updates to evicted state are dropped ensure that the requested upquery response is processed before any update for the evicted state.

Recursive upqueries of branching subgraphs, such as joins, are more complex. A join operator must emit a single correct response for each upquery it receives, even if it must make one or more recursive upqueries of its own to produce the needed state. Combining the upqueries' results directly would be incorrect: those upqueries execute independently, and updates can arrive between their responses. Joins thus issue recursive upqueries, but compute the final result exclusively with *join* upqueries once the recursive upqueries complete (multiple rounds of recursive upqueries may be required). These join upqueries execute within a single operator chain and exclude concurrent updates. Noria supports other branching operators, such as unions, which obey the same rules as joins.

Finally, a join upquery performed during update processing may encounter evicted state. In this case, Noria chooses to drop the update and evict dependent entries downstream; Noria statically analyzes the graph to compute the required eviction notices. There is a trade-off here: computing the missing entry could avoid future upqueries. Noria chooses to evict to avoid blocking the write path while filling in the missing state.

Such evictions are rare, but they can occur. For example, imagine a version of Figure 2 that adds `AuthorVotes`, which aggregates `VoteCount` by `stories.author`, and the following system state:

- `stories[id=1]` has `author=Elena`.
- `VoteCount[story_id=1]` has `vcount=8`.
- `AuthorVotes[author=Elena]` has `vcount=8`.
- `stories[id=2]` has `author=Bob`.
- `VoteCount[story_id=2]` is evicted.

Now imagine that an update changes story 2's author to Elena. When this update arrives at the join for `AuthorVotes`, that join operator upqueries for `VoteCount[story_id=2]`, which is evicted. As a result,

Noria sends an eviction notice for Elena—whose number of votes has changed—to `AuthorVotes`.

4.5 Partial and full state

Noria makes state partial whenever it can service upqueries using efficient index lookups. If Noria would have to scan the full state of an upstream operator to satisfy upqueries, Noria disables partial state for that operator. This may happen because every downstream record depends on all upstream ones—consider *e.g.*, the top 20 stories by vote count. In addition, the descendant eviction invariant implies that partial-state operators cannot have full-state descendants.

Partial-state operators in Noria start out fully evicted and are gradually and lazily populated by upqueries. As we show next, this choice has important consequences for Noria's ability to transition the data-flow efficiently.

5 Dynamic data-flow

Application queries evolve over time, so Noria's dynamic data-flow represents a continuously-changing set of SQL expressions. Existing data-flow systems run separate data-flows for each expression, initialize new operators with empty state and reflect only new writes, or require restarting from a checkpoint. Changes to the Noria program instead adapt the data-flow dynamically.

Given new or removed expressions, Noria *transitions* the data-flow to reflect the changes. Noria first plans the transition, reusing operators and state of existing expressions where possible (§5.1). It then incrementally applies these changes to the data-flow, taking care to maintain its correctness invariants (§5.2). Once both steps complete, the application can use new tables and queries.

The key challenges for transitions are to avoid unnecessary state duplication and to continue processing reads and writes throughout. Operator reuse and partial state help Noria address these challenges.

5.1 Determining data-flow changes

To initiate a transition, the application provides Noria with sets of added and removed expressions. Noria then computes required changes to the currently-running data-flow. This process resembles traditional database query planning, but produces a long-term *joint* data-flow across *all* expressions in the Noria program. This allows Noria to reuse existing operators for efficiency: if two queries include the same join, the data-flow contains it only once.

To plan a transition, Noria first translates each new expression into an extended *query graph* [21]. The query graph contains a node for each table or view in the expression, and an edge for every join or group-by clause. Noria uses query graphs to inexpensively reject many expressions from consideration [21, §3.4, 78, §3] and to quickly establish a set of *sharing candidates* for each new expression. The sharing candidates are existing expressions that likely overlap with the new expression. Next, Noria generates a verbose intermediate representation (IR), which splits the new expression into more fine-grained operators. This simplifies common subexpression detection, and allows Noria to efficiently merge the new IR with the cached IR of the sharing candidates.

For each sharing candidate, Noria reorders joins in the new IR to match the candidate when possible to maximize re-use opportunities. It then traverses the candidate's IR in topological order from the base tables. For each operator, Noria searches for a matching operator (or clique of operators) in the new IR. A match represents a reusable subexpression, and Noria splices the two IRs together at the deepest matches.

This process continues until Noria has considered all identified reuse candidates, producing a final, merged IR.

5.2 Data-flow transition

The combined final IRs of all current expressions represent the transition's *target* data-flow. Noria must add any operator in the final IR that does not already exist in the data-flow. To do so, Noria first informs existing operators of index obligations (§3.3) incurred by new operators that they must construct indexes for. Noria then walks the target data-flow in topological order and inserts each new operator into the running data-flow and bootstraps its state. Finally, after installing new operators and deleting removed queries' external views, Noria removes obsolete operators and state from the data-flow.

Bootstrapping operator state. When Noria adds a new stateful operator, it must ensure that the operator starts with the correct state. Partially-stateful operators and views start processing immediately. They are initially empty and bootstrap via upqueries in response to application reads during normal operation, amortizing the bootstrapping work over time. Fully-stateful operators are initially marked as "inactive", which causes them to ignore all incoming updates. Noria then executes a special, large upquery for *all* keys on behalf of the fully-stateful operator. Once the last upquery response has arrived, Noria activates the operator for update processing and moves on to the next new operator.

Base table changes. As applications evolve, developers often add or remove base table columns [17]. This affects *existing* operators in the data-flow: new updates from the base table may now lack values that existing operators expect. Noria could rebuild the data-flow or transform the existing base table state to effect such a change, but this would be inefficient for large base tables. Instead, Noria base tables internally track all columns that have existed in the table's schema, including those that have been deleted. When a base table processes an application write, it automatically injects default values for missing

columns (but does not store them). This permits queries for different base table schemas to coexist in the data-flow graph, and makes most base table changes cheap.

6 Implementation

Our Noria prototype implementation consists of 45k lines of Rust and can operate both on a single server and across a cluster of servers. Applications interface with Noria either through native Rust bindings, using JSON over HTTP, or through a MySQL protocol adapter.

6.1 Persistent data storage

Noria persists base tables in RocksDB [66], a high-performance key-value store based on log-structured merge (LSM) trees. Batches of application updates are synchronously flushed into RocksDB's log before Noria acknowledges them and admits them into the data-flow; a background thread asynchronously merges log entries into the LSM trees. Each base table index forms a RocksDB "column family". For base tables with non-unique indexes, Noria uses RocksDB's ordered iterators to efficiently retrieve all rows for an index key [14, 67].

Persistence reduces Noria's write throughput by about 5% over in-memory base tables. Reads are not greatly impacted when an application's working set fits in memory: only occasional upqueries access RocksDB, and these add $<$ 1ms of additional latency on a fast SSD.

6.2 Parallel processing

Noria shards the data-flow and allows concurrent reads and writes with minimal synchronization for parallelism.

Sharding. Noria processes updates in parallel on a cluster by hash-partitioning each operator on a key and assigning shards to different servers. Each machine runs a Noria *instance*, a process that contains a complete copy of the data-flow graph, but holds state only for its shards of each operator. When an operator with one hash partitioning links to an operator with a different partitioning, Noria inserts "shuffle" operators that perform inter-shard transfers over TCP connections. Upqueries across shuffle operators are expensive since they must contact all ancestor shards. This limits scalability, but allows operators below a shuffle to maintain partial state.

Multicore parallelism. Noria achieves multicore parallelism within each server in two ways: a server can handle multiple shards by running multiple Noria instances, and each instance runs multiple threads to process its shard. Each instance has two thread pools: *data-flow workers* process updates within the data-flow graph, and *read handlers* handle reads from external views.

At most one data-flow worker executes updates for each data-flow operator at a time. This arrangement yields CPU parallelism among different operators, and also allows lock-free processing within each operator. There are typically fewer data-flow workers than operators in the data-flow graph, so Noria multiplexes operator work across the worker threads. Within one instance, Noria schedules chains of operators with the same key as a unit. This reduces queueing and inter-core data movement at operator boundaries. It also allows Noria to optimize some upqueries: an upquery within a chain can simply access the ancestor's data synchronously, without worry of contamination from in-flight updates (§4.3).

Read handlers process clients' RPCs to read from external views. They must access the view with low latency and high concurrency, even while a data-flow worker is applying updates to the view. To minimize synchronization, Noria uses double-buffered hash tables for external views [27]: the data-flow worker updates one table while read handlers read the other, and an atomic pointer swap exposes new writes. This trades space and timeliness for performance: with skewed key popularity distributions, it can improve read throughput by 10$\times$ over a single-buffered hash table with bucket-level locks.

6.3 Distributed operation

A Noria controller process manages distributed instances on a cluster of servers, and informs them of changes to the data-flow graph and of shard assignments. Noria elects the controller and persists its state via ZooKeeper [34]. Clients discover the controller via ZooKeeper, and obtain long-lived read and write handles to send requests directly to instances.

Noria handles failures by rebuilding the data-flow. If the controller fails, Noria elects a new controller that restores the data-flow graph. It then streams the persistent base table data from RocksDB to rebuild fully-stateful operators and views. Partial operators are instead populated through on-demand upqueries. If individual instances fail, Noria rebuilds only the affected operators.

6.4 MySQL adapter

Our prototype includes an implementation of the MySQL binary protocol in a dedicated stateless adapter that appears as a standard MySQL server to the application. This adapter allows developers to easily run existing applications on Noria. The adapter transparently translates prepared statements and ad-hoc queries into transitions on Noria's data-flow, and applies reads and writes using Noria's API behind the scenes. Its SQL support is sufficiently complete to run some unmodified web applications (*e.g.*, JConf [74] written in Django [22]), and to run Lobsters with minimal syntax adaptation.

6.5 Limitations

Our current prototype has some limitations that we plan to address in future work; none of them are fundamental. First, it only shards by hash partitioning on a single column, and resharding requires sending updates through

a single instance, which limits scalability. Second, it re-computes data-flow state on failure; recovering from snapshots or data-flow replicas would be more efficient (*e.g.*, using selective rollback [35]). And third, it does not currently support range indices or multi-column joins.

7 Applications

This section discusses our experiences with developing Noria applications. Noria aims to simplify the development of high-performance web applications; several aspects of our implementation help it achieve that goal.

First, applications written for a MySQL database can use Noria directly via its MySQL adapter, provided they generate parameterized SQL queries (for instance, via libraries like PHP Data Objects [69] or Python's MySQL connector [55, §10.6.8]). Porting typically proceeds in three steps. First, the developer points the application at the Noria MySQL adapter instead of a MySQL server and imports existing data into Noria from database dumps. The application will immediately see performance improvements for read queries that formerly ran substantial in-line compute. Though the MySQL adapter even supports ad-hoc read queries (it transitions the data-flow as required to support each query), the most benefit will be seen for frequently-reused queries. Second, the developer creates views for computations that the MySQL application manually materialized, such as the per-story vote count in Lobsters. These views coexist with the manual materializations, and allow existing queries to continue to work as the developer updates the write path so that it no longer manually updates derived views and caches. Third, the developer incrementally rewrites their application to rely on natural views and remove manual write optimizations. These changes gradually increase application performance as the developer removes now-unnecessary complexity from the application's read and write paths.

The porting process is not burdensome. We ported a PHP web application for college room ballots—developed by one of the authors and used production for a decade—to Noria; the process took two evenings, and required changes to four queries. We also used the MySQL adapter to port the Lobsters application's queries to Noria; the result is a focus of our evaluation.

Developing native Noria applications can be even easier. We developed a simple web application to show the results of our continuous integration (CI) tests for Noria. The CI system stores its results in Noria, and the web application displays performance results and aggregate statistics. Since we developed directly for Noria, we were not tempted to cache intermediate results or apply other manual optimizations, and could use aggregations and joins in queries without fear that performance would suffer as a result (*e.g.*, due to aggregations over the long commit history). Most application updates reduced to single-table inserts, deletes, or updates.

Limitations. Though applications traditionally use parameterized queries to avoid SQL injection attacks and cache query plans, Noria parameterized queries also build materialized views. An application with many distinct parameterized queries can thus end up with more views than necessary. The developer can correct this by adding shared views. Our prototype does not yet support update and delete operations conditioned on non-primary key columns, and lacks support for parameterized range queries (*e.g.*, `age > ?`), which some applications need. Planned support for range indexes and an extended base table implementation will address these limitations.

8 Evaluation

We evaluated our Noria prototype using backend workloads generated from the production Lobsters web application, as well as using individual queries. Our experiments seek to answer the following questions:

1. What performance gains does Noria deliver for a typical database-backed web application? (§8.1)
2. How does Noria perform compared to a MySQL/memcached stack, the materialized views of a commercial database, and an idealized cache-only deployment? (§8.2)
3. Given a scalable workload, how does our prototype utilize multiple servers, and how does it compare to a state-of-the-art data-flow system? (§8.3)
4. What space overhead does Noria's data-flow state impose, and how does Noria perform with limited memory and partial state? (§8.4)
5. Can Noria data-flows adapt to new queries and input schema changes without downtime? (§8.5)

Setup. In all experiments, Noria and other storage backends run on an Amazon EC2 c5.4xlarge instance with 16 vCPUs; clients run on separate c5.4xlarge instances unless stated otherwise. Our setup is "partially open-loop": clients generate load according to a Poisson distribution of interarrival-times and have a limited number of backend requests outstanding, queueing additional requests. This ensures that clients maintain the measurement frequency even during periods of high latency [45]. Our test harness measures offered request throughput and "sojourn time" [62], which is the delay from request generation until a response returns from the backend.

8.1 Application performance: Lobsters

We first evaluate Noria's performance on a realistic web application workload to answer two questions:

1. Do Noria's fast reads help it outperform a conventional database on a real application workload, even on a hand-optimized application?

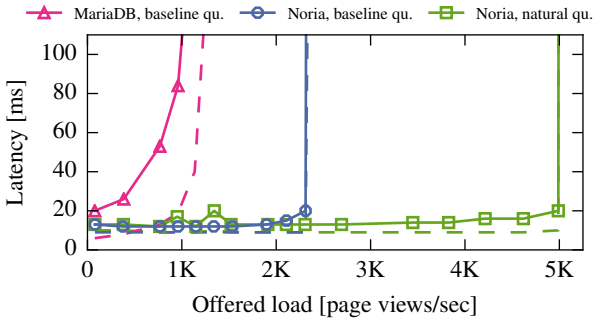


Figure 6: Noria scales Lobsters to a 5× higher load than MariaDB (2.3× with baseline queries) at sub-100ms 95%ile latency (dashed: median). MariaDB is limited by read computation, while Noria becomes write-bound.

2. Can Noria preserve good performance for an application without hand optimization?

Our workload models production Lobsters traffic. The benchmark emulates authenticated Lobsters users visiting different pages according to the access frequencies and popularity distributions in the production workload [32]. Lobsters is a Ruby-on-Rails application, but our benchmark generates database operations directly in order to eliminate Rails overhead. We seed the database with 9.2k users, 40k stories and 120k comments—the size of the real Lobsters deployment—and run increasing request loads to push the different setups to their limits.

The baseline queries include the Lobsters developers' optimizations, which manually materialize and maintain aggregate values like vote counts to reduce read-side work. We also developed "natural" queries that produce the same results using Noria data-flow to compute aggregations rather than manual optimizations. We compare MariaDB (a community-developed MySQL fork; v10.1.34) with Noria using baseline queries, and then to Noria using natural queries (both via Noria's MySQL adapter). We configured MariaDB to use a thread pool, to avoid flushing to disk after transactions, and to store the database on a ramdisk to remove overheads unrelated to query execution. With the baseline queries, the median page view executes 11 queries; this reduces to eight with natural queries. This experiment uses an m5.24xlarge EC2 instance for the CPU-intensive clients.

Figure 6 shows the results as throughput-latency curves. An ideal system would show as a horizontal line with low latency; in reality, each setup hits a "hockey stick" once it fails to keep up with the offered load. MariaDB scales to 1,000 pages/second, after which it saturates all 16 CPU cores with read-side computation (*e.g.*, for per-page notification counts [33]). Noria running the same baseline queries scales to a 2.3× higher offered load, since its incremental write-side processing avoids redundant re-computation on reads.

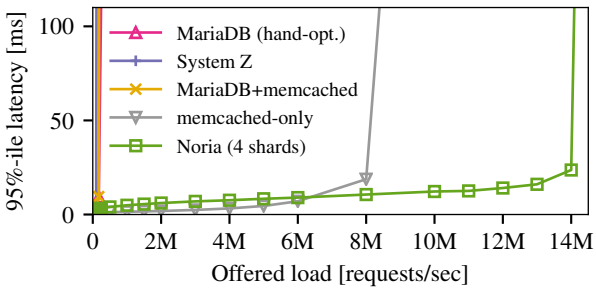
The baseline queries manually pre-compute aggregates. MariaDB requires this for performance: without the pre-computation, it supports just 20 pages/sec. Noria instead maintains pre-computed aggregates in its data-flow. This allows us to include the aggregations directly in the queries, which normalizes the base table schema, reduces write load, and avoids bugs due to missed updates to pre-computed values. With all aggregate computation moved into Noria's data-flow ("natural queries"), throughput scales higher still, to 5,000 pages/second (5× MariaDB). Eliminating application pre-computation reduces overall write load and compacts the data-flow, which lets Noria parallelize it more effectively.

The result is that Noria achieves both good performance and natural, robust queries. We observed similar benefits with other applications (*e.g.*, a synthetic TPC-W-like workload), which we omit for space.

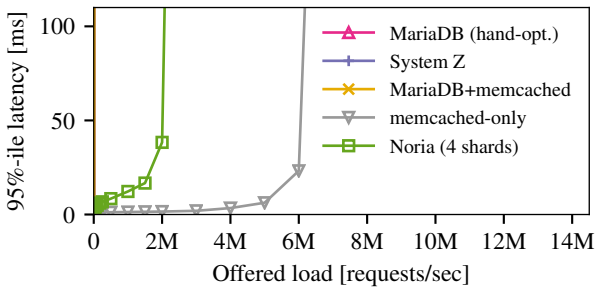
8.2 In-depth performance comparison

We compare to alternative systems using a subset of Lobsters. This restriction gives us better control over workload properties, while capturing the aspects of web workloads that motivated the Noria design. We use one kind of write, inserting a vote, and one read query, `StoriesWithVC` from Figure 2. This read query fetches stories and their vote counts; 85% of page views in production Lobsters are for pages that execute this query.

We compare five single-server deployments that all have access to the same resources, but differ in how they store and calculate the per-story vote count. **MariaDB** uses the baseline Lobsters approach of pre-computing and storing vote counts in a column of the Lobsters `stories` table. **System Z**, a commercial database with materialized view support, uses an incrementally-maintained materialized view defined similarly to `StoriesWithVC`; we use System Z to compare database view maintenance with Noria's data-flow-based approach. MariaDB and System Z run at the fastest transactional isolation level ("read uncommitted") and are configured to keep data in memory. **MariaDB+memcached** adds a demand-filled memcached (v1.5.6) cache [54] to MariaDB that caches `StoryWithVC` entries. This reduces read load on MariaDB, but complicates application code even beyond pre-computation: writes must invalidate the cache and reads must sometimes populate it. We also measure **memcached-only** without a relational backend. This setup offers good performance, but is unrealistic: it does not store individual votes or stories, is not persistent, and cannot prevent double-voting. It helps us estimate how a backend that serves all reads from memory and does minimal work for writes might perform. Finally, we measure **Noria** sharded four ways on `stories.id`, with the remaining 12 cores serving reads.



(a) Read-heavy workload (95%/5%): Noria outperforms all other systems (all but memcached at 100–200k requests/sec).



(b) Mixed read-write workload (50%/50%): Noria outperforms all systems but memcached (others are at 20k requests/sec).

Figure 7: A Lobsters subset (Figure 2) benchmarked on Noria hand-optimized MariaDB, System Z's materialized views, a MariaDB/memcached setup, and on memcached only, all with Zipf-distributed ($s = 1.08$) reads and votes.

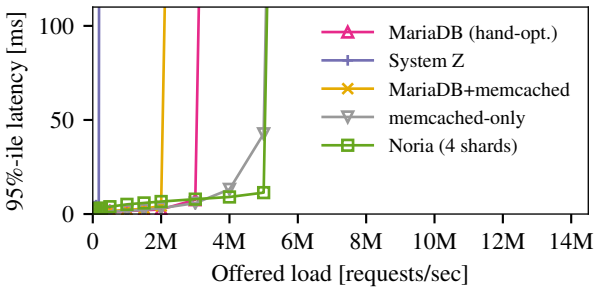


Figure 8: For a uniformly-distributed, read-heavy (95%/5%) workload on Figure 2, Noria performs similarly to the (unrealistic) memcached-only setup.

Noria uses natural queries; other systems except System Z manually pre-compute vote counts.

Clients read and insert votes for randomly-chosen stories; we measure the 95th-percentile latency for each offered load. Before measurement begins, we populate the `stories` table with 500k records and perform 40 seconds of warmup using the same workload as the benchmark itself. Absolute throughput is higher in these experiments because the data-flow only contains a single query and clients batch reads and writes for up to 1ms.

Figure 7 shows results for a **skewed workload** similar to Lobsters', with story popularity following a Zipfian distribution ($s = 1.08$). With 95% reads, Noria outperforms all other systems, including the unrealistic cache-only deployment (Figure 7a). Most updates write votes for popular stories, which creates write contention problems in MariaDB and System Z. The MariaDB+memcached setup performs equally poorly: on memcached invalidations for popular keys, multiple clients miss and a "thundering herd" of clients simultaneously issues database queries [54, §3.2.1]. memcached on its own scales, but Noria outperforms it (despite doing more work) since Noria's lockless views avoid contention for popular keys. Noria scales to 14M requests/second with four shards. Noria also handles a **write-heavy workload** (50% writes) well (Figure 7b): although absolute performance has dropped, Noria still outperforms all other systems apart from the cache-only setup. This is because sharding allows data-parallel write processing, which helps Noria scale to 2M requests/second.

With a (less-realistic) **uniform workload**, other systems come closer to Noria's 5M requests/second (Figure 8). System Z does better than before, but suffers from slow writes to the materialized view. MariaDB+memcached, perhaps surprisingly, performs worse than MariaDB, which scales to 3M requests/second: the reason lies in the extra work (and RPCs) the application must perform for invalidations. This illustrates that a look-aside cache only helps if it avoid expensive queries; a write-through cache avoids invalidation overheads, but would still perform worse than the idealized memcached-only setup (and thus, than Noria).

Separately, we evaluated Noria's view maintenance against DBToaster [2, 53], a state-of-the-art materialized view maintenance system that compiles view definitions to native code. DBToaster (v2.2.3387) lacks support for persistent base tables, concurrent reads, or multicore parallelism—its only read operation snapshots entire views—but it does provide fast updates to materialized views. When we constrain Noria to only one shard and data-flow worker thread, we expect DBToaster to outperform it, since DBToaster's generated C++ code does close-to-minimal work to incrementally maintain the vote count. We measure the write throughput of 50M uniformly-distributed votes that update `StoriesWithVC` for 500k stories. Noria achieves 240k single-record writes/second for fully-populated state, and 1M writes/second for fully-evicted state. DBToaster only supports fully-populated state, and achieves 520k single-record writes/second. At the same time, Noria is more memory-efficient, using 6.2 GB of memory for base tables and all derived state, 36% of DBToaster's 17 GB.

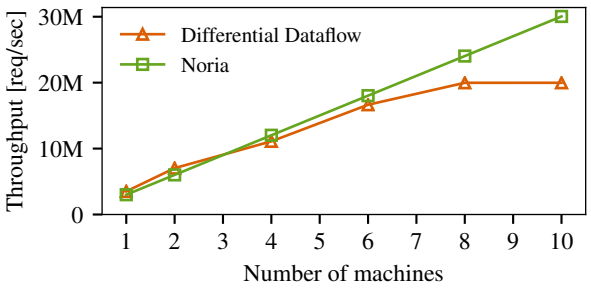


Figure 9: For a uniform 95%/5% workload, Noria scales to ten machines with sub-100ms 95th %tile latency by sharding the data-flow. Differential dataflow [44] scales less well due to its inter-worker coordination.

Additionally, Noria can process shards in parallel and use more machines to increase throughput.

8.3 Distribution over multiple servers

We next evaluate Noria's support for distributed operation. Can Noria effectively use multiple machines' resources given a scalable workload?

We evaluate the 95%-read Lobsters subset from §8.2 with two million stories. We shard the data-flow on `stories.id` and vary the number of machines from one to ten, with each machine hosting four shards. For a deployment with n Noria machines, we scale client load to $n\times$3M requests/second in a partially open-loop test harness. This arrangement achieves close to Noria's maximum load at sub-100ms 95th-percentile latency for two million stories on one machine. Load generators select stories uniformly at random, so the workload is perfectly shardable. The ideal result is a straight diagonal, with n machines achieving n times the throughput of a single one. Figure 9 shows that Noria achieves this and serves the full per-machine load at all points.

We also implemented this benchmark for a state-of-the-art Differential Dataflow (DD) implementation (v0.7) in Rust [44] based on Naiad and its earlier version of DD [46, 51]. Since DD lacks a client-facing RPC interface, we co-locate DD clients with workers; this does not disadvantage DD since load generation is cheap compared to RPC processing. DD uses 12 worker threads and four network threads per machine.

Figure 9 shows that Noria is competitive with DD on this benchmark. On one and two machines, DD supports a slightly higher per-machine load (3.5M requests/second *vs.* Noria's 3M) within our 95th-percentile latency budget of 100ms. Beyond four machines, however, DD fails to meet Noria's maximum per-machine load. Its supported throughput tails off to around 20M requests/sec at ten machines. This tail-off is due to DD's progress-tracking protocol, which coordinates between workers to expose writes atomically, and which imposes increasing overhead as the number of machines grows. DD amortizes this coordination by increasing its batch size, and consequently sees increased latency as throughput increases. Noria avoids such coordination and scales well, but offers only eventually-consistent reads.

8.4 State size

Noria relies on partial state to keep its memory footprint low. How much of Noria's state for Lobsters can be partial, and how does Noria perform when it evicts from partial state to meet a memory limit? We investigate these questions using the full Lobsters application, first at Lobsters production scale, and then at 10× scale.

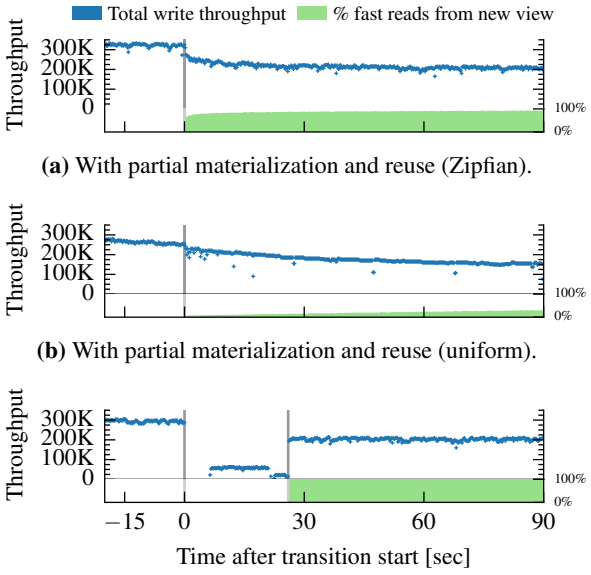
The Noria data-flow for the natural Lobsters queries has 235 operators, of which 60 of are stateful. With partial state disabled, *i.e.*, forcing all data-flow operators to keep full state, Noria needs 789 MB of in-memory state (8× the base table size of 137 MB). With partial state enabled, 35 of the stateful operators can use partial state; the remaining 25 are part of unparameterized views (*e.g.*, all stories on the front page) whose state Noria cannot make partial as they lack suitable keys. Together, the non-partial state occupies 73 MB: Noria's essential memory requirement for Lobsters therefore amounts to 9% of total state (adding an overhead of 53% of base table size). Noria can evict and re-compute the remaining 91% of state should it exceed a memory limit.

As for any cache, this memory limit should exceed the application's working set size to achieve low read latency and avoid thrashing of evictions and upqueries. For Lobsters, the working set size depends on the offered load, as higher load means a wider range of stories are read. We determine it by varying Noria's state size limit (and hence, eviction frequency) and measuring 95th-percentile read latency. With production-scale Lobsters data, Noria's working set contains 525 MB of state (60% of total, 3.8× base tables) at an offered load of 2,300 pages/second. However, with a few thousand users, the production Lobsters deployment is small. Our benchmark further understates its size as we use synthetic story and comment texts of a few bytes. Hence, we repeated this experiment with the Lobsters data scaled up by 10×. Noria meets sub-100ms 95th percentile latency at 2,300 pages/second if the memory limit exceeds the 2.6 GB working set (38% of 7 GB total state; 3× base tables).

These results suggest that Noria imposes a reasonable space overhead (around 3× base table size) for Lobsters, and that partial state is key to reducing the overhead.

8.5 Live data-flow adaptation

In a traditional database, query changes are easy and instantaneous. Can Noria's data-flow adaptation seamlessly transition to include new SQL expressions? The goal is for the transition to complete quickly, for write



(a) With partial materialization and reuse (Zipfian).

(b) With partial materialization and reuse (uniform).

(c) No reuse or partial materialization (Zipfian).

Figure 10: Reuse and partial state allow Noria to adapt the live data-flow. Gray lines delimit start and end of the transition (in (a) and (b), the transitions are almost instantaneous); the green shaded area shows the fraction of new view reads that require no upqueries. Reads from the old view (not shown) proceed at full speed throughout.

performance to remain stable, for reads from existing views to be unaffected, and for reads from newly-added views to quickly achieve low latency.

We test this by adding a modified version of the `StoriesWithVC` view to the Lobsters subset. This new view, `StoriesWithRatings`, uses numeric ratings stored in a `ratings` base table instead of votes. It also reflects old votes scaled to a rating. We first load an unsharded Noria with 2M stories and 30M votes, then transition to the new program. Once the transition finishes, clients perform "rating reads" from `StoriesWithRatings` and start writing to the new `ratings` table. Throughout the experiment, clients also read the `StoriesWithVC` view, and write to the `votes` table. We expect post-transition throughput to be reduced—the new data-flow graph is larger, with more tables and deeper paths—although removing the old view would increase throughput again. However, we hope that throughput and latency do not suffer greatly *during* the transition.

Figure 10a shows the transition with reuse and partial materialization enabled. The transition completes immediately: Noria creates the new operators and view as empty, and populates them on demand in response to reads. Due to the skewed read and write distributions, upqueries for only a few popular keys suffice for Noria to serve the majority of rating reads without recursive upqueries. Reuse is also crucial: without reusing `VoteCount`, Noria must upquery rating reads by recomputing from the base tables. This leads to slow upqueries for popular stories, as the data-flow must recount their votes. With reuse enabled, pre-computed vote counts satisfy the upqueries. The results also follow this pattern for a uniform workload (Figure 10b). Initially, most rating reads are slow, but fast reads increase as the partial state populates; write throughput is reduced because data-flow updates contend with upquery responses. Contention increases as more entries populate, since fewer updates hit evicted state.

Figure 10c shows the same transition (with a Zipfian workload), but with partial materialization and operator reuse disabled. Noria fully populates the `StoriesWithRatings` view and all internal stateful operators during the transition. It copies `votes` and `stories` to bootstrap the rating aggregation state, and then copies the resulting state again to initialize the new external view. Each copy stops write processing for several seconds, and Noria's state transfer to the new operators via the data-flow slows down concurrent writes. When transition completes after 25 seconds, the `StoriesWithRatings` view is fully materialized and all rating reads are fast. This illustrates that partial state and reuse are crucial for downtime-free data-flow transitions.

How often can Noria achieve a live transition in practice? In a separate analysis of query and schema changes in HotCRP and TPC-W, we found that Noria live-transitioned for over 95% of program changes. Existing approaches are less flexible: System Z must rebuild its materialized views on change; a memcached cluster must be carefully transitioned [54, §4.3]; DBToaster lacks support for query changes; and even relational databases pause writes during some schema updates.

8.6 Discussion

We evaluated Lobsters both at production scale and at 10× scale, but many web applications are much larger still. We believe that Noria can also support such applications. For applications with many queries, and consequently a large data-flow, Noria can assign shards of only some operators to each machine, sending cross-operator traffic over the network. Similarly, Noria can shard large base tables and operators with large state across machines. Efficient resharding and partitioning the data-flow to minimize network transfers are important future work for Noria to achieve truly large scale.

We also believe Noria is well suited for applications whose working sets change over time. Many large, real-world applications see such changing workloads; for instance, an old story may suddenly become popular. As

clients request such items, Noria's upqueries bring them into the working set, making subsequent reads fast.

9 Related work

Noria builds on considerable related work.

Data-flow systems excel at data-parallel computing [36, 51], including on streams, but cannot serve web applications directly. They only achieve low-latency incremental updates at the expense of windowed state (and incomplete results) or by keeping full state in memory. Noria's partially-stateful data-flow lifts this restriction. A few data-flow systems can reuse operators automatically: for example, Nectar [28] detects similar subexpressions in DryadLINQ programs, similar to Noria's automated operator reuse, using DryadLINQ-specific merge and rewrite rules. Support for dynamic changes to a running data-flow is more common: CIEL [52] dynamically extends batch-processing data-flows, as does Ray [58] for stateful "actor" operators' state transitions in reinforcement learning applications. Noria dynamically changes long-running, low-latency streaming computations by modifying the data-flow; unlike existing streaming data-flow systems like Naiad [51] or Spark Streaming [76], it has no need for a restart or recovery from a checkpoint.

Stream processing systems [3, 11, 39, 71, 76] often use data-flow, but usually have windowed state and static queries that process only new records. STREAM [6] identifies opportunities for operator reuse among static queries; Noria achieves similar reuse for dynamic queries. S-Store [47] lacks Noria's partial materialization and state reuse, but combines a classic database with a stream processing system using trigger-based view maintenance. S-Store enables transactional processing, a future goal for Noria.

Database **materialized views** [29, 41] were devised to cache expensive analytical query results. Commercial databases' materialized view support [1] is limited [49, 63] and views must usually be rebuilt on change. However, there is considerable research on incremental view maintenance in databases [30, 40, 41, 70, 77, 81]. Noria builds upon ideas from this work, but applies them in the context of a concurrent, stateful data-flow system for web applications. This requires efficient fine-grained access to views, solutions to new coordination problems and concurrency races, as well as inexpensive long-term adaptation as view definitions change. DBToaster [2, 53] supports incremental view maintenance under high write loads with generated recursive delta query implementations. Noria sees lower single-threaded performance, but supports parallel processing and changing queries; adding native-code generation to Noria might further improve its performance, but would complicate operator reuse. Pequod [37] and DBProxy [4] support **partial materialization** in response to client demand, although Pequod is limited to static queries, and unlike Noria, neither shares state nor processing across queries.

The problem of detecting shared subexpressions (§5.1) is a **multi-query optimization** (MQO) problem [21, 59, 78]. MQO tries to maximize sharing across a batch of expressions, with the freedom to rewrite any expression to suit the others. Like joint query processing systems [10, 25, 31], Noria faces the more restricted problem of mutating *new* expressions to increase their opportunity to share *existing* expressions in the data-flow.

A wide array of tools deal with websites' **query and schema transitions** [9, 23, 26, 56, 65]. Like Noria, they aim to transition backend stores without interruption in client service, but they require developers to manually configure complex "ghost tables" or binlog-following triggers. Base table schema changes increase complexity further [73]. Noria handles query changes transparently, and efficiently applies common base table schema changes by supporting many concurrent base table schemas. Most of its data-flow transitions are live for reads and writes without added complexity.

Finally, some open-source systems have experimented with flexible query and schema changes. Apache Kafka [5] achieves some flexibility in query and schema changes as used by the New York Times [68], and similar ideas were proposed as an extension proposal for Samza [38]. To our knowledge, however, no prior system achieves the performance and flexibility of Noria.

10 Conclusions

Noria is a web application backend that delivers high performance while allowing for simplified application logic. Partially-stateful data-flow is essential to achieving this goal: it allows fast reads, restricts Noria's memory footprint to state that is actually used, and enables live changes to the data-flow. In future work, we plan to add more flexible sharding, range indexes, and better eviction strategies.

Noria is open-source software and available at:

https://pdos.csail.mit.edu/noria

Acknowledgements

We thank Joana da Trindade and Nikhil Benesch for contributions to our implementation, as well as Frank McSherry for assisting with implementation and tuning of the differential dataflow benchmark. Jon Howell provided helpful feedback that much improved the paper, as did Ionel Gog, Frank McSherry, David DeWitt, Sam Madden, Amy Ousterhout, Tej Chajed, Anish Athalye, and the PDOS and Database groups at MIT. We are also grateful to the helpful comments we received from our anonymous reviewers, as well as from Wyatt Lloyd, our shepherd. This work was funded through NSF awards CSR-1301934, CSR-1704172, and CSR-1704376.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt, Sept. 2000, pages 496–505.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pages 968–979.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pages 1033–1044.

[4] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. "DBProxy: a dynamic data cache for web applications". In: *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. Mar. 2003, pages 821–831.

[5] Apache Software Foundation. *Apache Kafka: a distributed streaming platform*. URL: http://kafka.apache.org/ (visited on 09/14/2017).

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. "STREAM: The Stanford Data Stream Management System". In: *Data Stream Management: Processing High-Speed Data Streams*. Edited by Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Berlin/Heidelberg, Germany: Springer, 2016, pages 317–336.

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. "Finding a Needle in Haystack: Facebook's Photo Storage". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pages 1–8.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the USENIX Annual Technical Conference*. San Jose, California, USA, June 2013, pages 49–60.

[9] Mark Callaghan. *Online Schema Change for MySQL*. URL: https://www.facebook.com/note.php?note_id=430801045932 (visited on 02/01/2017).

[10] George Candea, Neoklis Polyzotis, and Radek Vingralek. "A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses". In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pages 277–288.

[11] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. "Apache Flink: Stream and batch processing in a single engine". In: *IEEE Data Engineering* 38.4 (Dec. 2015).

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006.

[13] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. "Realtime Data Processing at Facebook". In: *Proceedings of the 2016 SIGMOD International Conference on Management of Data*. San Francisco, California, USA, 2016, pages 1087–1098.

[14] CockroachDB. *Structured data encoding in CockroachDB SQL*. Jan. 2018. URL: https://github.com/cockroachdb/cockroach/blob/master/docs/tech-notes/encoding.md (visited on 04/20/2018).

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pages 1277–1288.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013), 8:1–8:22.

[17] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. "Schema Evolution in Wikipedia: toward a Web Information System Benchmark". In: *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*. June 2008.

[18] Databricks, Inc. *Structured Streaming in Production – Recover after changes in a streaming query*. URL: https://docs.databricks.com/spark/latest/structured-streaming/production.html#recover-after-changes-in-a-streaming-query (visited on 09/06/2018).

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, Oct. 2007, pages 205–220.

[20] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. "Development and Deployment at Facebook". In: *IEEE Internet Computing* 17.4 (July 2013), pages 8–17.

[21] Sheldon Finkelstein. "Common Expression Analysis in Database Applications". In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. Orlando, Florida, USA, June 1982, pages 235–245.

[22] Django Software Foundation. *Django: The Web framework for perfectionists with deadlines*. Mar. 2018. URL: https://www.djangoproject.com/ (visited on 03/20/2018).

[23] Matt Freels. *TableMigrator*. URL: https://github.com/freels/table_migrator (visited on 02/01/2017).

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, USA, Oct. 2003, pages 29–43.

[25] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. "SharedDB: Killing One Thousand Queries with One Stone". In: *Proceedings of the VLDB Endowment* 5.6 (Feb. 2012), pages 526–537.

[26] GitHub, Inc. *gh-ost: GitHub's online schema migration for MySQL*. URL: https://github.com/github/gh-ost (visited on 02/01/2017).

[27] Jon Gjengset. *evmap: A lock-free, eventually consistent, concurrent multi-value map*. URL: https://github.com/jonhoo/rust-evmap (visited on 09/13/2018).

[28] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 75–88.

[29] Himanshu Gupta and Inderpal Singh Mumick. "Selection of views to materialize in a data warehouse". In: *IEEE Transactions on Knowledge and Data Engineering* 17.1 (Jan. 2005), pages 24–43.

[30] Himanshu Gupta and Inderpal Singh Mumick. "Incremental Maintenance of Aggregate and Outerjoin Expressions". In: *Information Systems* 31.6 (Sept. 2006), pages 435–464.

[31] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. Baltimore, Maryland, USA, June 2005, pages 383–394.

[32] Peter Bhat Harkins. *Lobste.rs access pattern statistics for research purposes*. Mar. 2018. URL: https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b (visited on 03/12/2018).

[33] Peter Bhat Harkins. *replying_comments view in Lobsters*. Feb. 2018. URL: https://github.com/lobsters/lobsters/blob/640f2cdca10cc737aa627dbdf0bbe398b81b497f/db/views/replying_comments_v06.sql (visited on 04/20/2018).

[34] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Boston, Massachusetts, USA, June 2010, pages 149–158.

[35] Michael Isard and Martín Abadi. "Falkirk Wheel: Rollback Recovery for Dataflow Systems". In: *CoRR* abs/1503.08877 (2015).

[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pages 59–72.

[37] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. "Easy Freshness with Pequod Cache Joins". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, Washington, USA, Apr. 2014, pages 415–428.

[38] Martin Kleppmann. *Turning the database inside-out with Apache Samza*. Mar. 2015. URL: https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html (visited on 05/09/2016).

[39] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia, May 2015, pages 239–250.

[40] Per-Åke Larson and Jingren Zhou. "Efficient Maintenance of Materialized Outer-Join Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Apr. 2007, pages 56–65.

[41] Ki Yong Lee and Myoung Ho Kim. "Optimizing the Incremental Maintenance of Multiple Join Views". In: *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. Bremen, Germany, Nov. 2005, pages 107–113.

[42] Lobsters Developers. *Lobsters Database Schema (schema.rb)*. Apr. 2018. URL: https://github.com/lobsters/lobsters/blob/93fe0fdd74028cf678134d6d112ae084d8fdd928/db/schema.rb#L145-L148 (visited on 04/23/2018).

[43] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: https://lobste.rs (visited on 03/02/2018).

[44] Frank McSherry. *Differential Dataflow in Rust*. URL: https://crates.io/crates/differential-dataflow (visited on 01/15/2017).

[45] Frank McSherry. *Throughput and Latency in Differential Dataflow: open-loop measurements*. Aug. 2017. URL: https://github.com/frankmcsherry/blog/blob/master/posts/2017-07-24.md#addendum-open-loop-measurements-2017-08-14 (visited on 04/13/2018).

[46] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.

[47] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. "S-Store: Streaming Meets Transaction Processing". In: *Proceedings of the VLDB Endowment* 8.13 (Sept. 2015), pages 2134–2145.

[48] Jhonny Mertz and Ingrid Nunes. "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches". In: *ACM Computing Surveys* 50.6 (Nov. 2017), 98:1–98:34.

[49] Microsoft, Inc. *Create Indexed Views – Additional Requirements*. SQL Server Documentation. URL: https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views#additional-requirements (visited on 04/16/2017).

[50] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. "f4: Facebook's Warm BLOB Storage System". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pages 383–398.

[51] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.

[52] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pages 113–126.

[53] Milos Nikolic, Mohammad Dashti, and Christoph Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates". In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pages 511–526.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398.

[55] Oracle Corp. *MySQL Connector/Python Developer Guide*. URL: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursorprepared.html (visited on 09/05/2018).

[56] Percona LLC. *pt-online-schema-change*. URL: https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html (visited on 02/01/2017).

[57] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. "Transactional Consistency and Automatic Management in an Application Data Cache". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 279–292.

[58] "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018.

[59] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, May 2000, pages 249–260.

[60] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. "How to Roll a Join: Asynchronous Incremental View Maintenance". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, 2000, pages 129–140.

[61] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. "Continuous Deployment at Facebook and OANDA". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas, USA, 2016, pages 21–30.

[62] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, 2006, pages 239–252.

[63] Jes Schultz Borland. *What You Can (and Can't) Do With Indexed Views*. Brent Ozar Unlimited Blog. URL: https://www.brentozar.com/archive/2013/11/what-you-can-and-cant-do-with-indexed-views/ (visited on 04/16/2017).

[64] Ziv Scully and Adam Chlipala. "A Program Optimization for Automatic Database Result Caching". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Paris, France, 2017, pages 271–284.

[65] SoundCloud Ltd. *Large Hadron Migrator*. URL: https://github.com/soundcloud/lhm (visited on 02/01/2017).

[66] Facebook Open Source. *A persistent key-value store for fast storage environments*. Apr. 2018. URL: http://rocksdb.org/ (visited on 04/20/2018).

[67] Facebook Open Source. *MyRocks data dictionary format*. Apr. 2018. URL: https://github.com/facebook/mysql-5.6/wiki/MyRocks-data-dictionary-format (visited on 04/20/2018).

[68] Boerge Svingen. *Publishing with Apache Kafka at The New York Times*. Confluent, Inc. blog. Sept. 2017. URL: https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/ (visited on 09/14/2017).

[69] The PHP Group. *PHP Data Objects*. URL: http://php.net/manual/en/book.pdo.php (visited on 09/05/2018).

[70] Frank W. Tompa and Joseph A. Blakeley. "Maintaining Materialized Views Without Accessing Base Data". In: *Information Systems* 13.4 (Oct. 1988), pages 393–406.

[71] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Snowbird, Utah, USA, June 2014, pages 147–156.

[72] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (Jan. 2009), pages 40–44.

[73] Jacqueline Xu. *Online migrations at scale*. Stripe engineering blog. URL: https://stripe.com/blog/online-migrations (visited on 02/01/2017).

[74] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-backed Applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, June 2016, pages 631–647.

[75] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pages 15–28.

[76] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 423–438.

[77] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. "Lazy Maintenance of Materialized Views". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. Vienna, Austria, Sept. 2007, pages 231–242.

[78] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. "Efficient Exploitation of Similar Subexpressions for Query Processing". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Beijing, China, 2007, pages 533–544.

[79] Jingren Zhou, Per-Åke Larson, and Jonathan Goldstein. *Partially Materialized Views*. Technical report MSR-TR-2005-77. Microsoft Research, June 2005.

[80] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. "Dynamic Materialized Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Istanbul, Turkey, Apr. 2007, pages 526–535.

[81] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. "View Maintenance in a Warehousing Environment". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. San Jose, California, USA, May 1995, pages 316–327.

Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!

Xingda Wei, Zhiyuan Dong, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: rongchen, haibochen@sjtu.edu.cn

Abstract

There is currently an active debate on which RDMA primitive (i.e., one-sided or two-sided) is optimal for distributed transactions. Such a debate has led to a number of optimizations based on one RDMA primitive, which was shown with better performance than the other.

In this paper, we perform a systematic comparison between different RDMA primitives with a combination of various optimizations using representative OLTP workloads. More specifically, we first implement and compare different RDMA primitives with existing and our new optimizations upon a single well-tuned execution framework. This gives us insights into the performance characteristics of different RDMA primitives. Then we investigate the implementation of optimistic concurrency control (OCC) by comparing different RDMA primitives using a phase-by-phase approach with various transactions from TPC-C, SmallBank, and TPC-E. Our results show that no single primitive (one-sided or two-sided) wins over the other on all phases. We further conduct an end-to-end comparison of prior designs on the same codebase and find none of them is optimal.

Based on the above studies, we build `DrTM+H`, a new hybrid distributed transaction system that always embraces the optimal RDMA primitives at each phase of transactional execution. Evaluations using popular OLTP workloads including TPC-C and SmallBank show that `DrTM+H` achieves over 7.3 and 90.4 million transactions per second on a 16-node RDMA-capable cluster (ConnectX-4) respectively, without locality assumption. This number outperforms the pure one-sided and two-sided systems by up to 1.89X and 2.96X for TPC-C with over 49% and 65% latency reduction. Further, `DrTM+H` scales well with a large number of connections on modern RDMA network.

1 Introduction

Distributed transactions with serializability and high availability provide a powerful abstraction to programmers with the illusion of a single machine that executes transactions with strong consistency and never fails. Although distributed transaction used to seem slow [19], the prevalence of fast networking features such as RDMA has boosted the performance of distributed transactions by orders of magnitudes [51, 5, 11, 18]. RDMA NIC (RNIC) provides high bandwidth, ultra-low latency datagram communication (two-sided primitive), together with offloading technology (one-sided primitive): the network card can directly access the memory of remote machines while bypassing kernel and remote CPUs.

Recently, there is an active debate over which RDMA primitive, namely one-sided or two-sided, is better suited for distributed transactions. One-sided primitive (e.g., `READ`, `WRITE`, and `ATOMIC`) provides higher performance and lower CPU utilization [10, 11, 51, 5]. On the other hand, two-sided primitive simplifies application programming and is less affected by hardware restrictions such as the limitation of RNIC's cache capacity [16, 18].

It is often challenging for system designers to choose the right primitive for transactions based on previous studies. Most work on RDMA-enabled transactions presents a new system built from scratch and compares its performance with previous ones using other codebases. Some only compare the performance of different primitives or designs using micro-benchmarks. This makes their results hard to interpret: differences in hardware configurations and software stacks affect the observable performance. Further, different RDMA primitives may significantly affect the overall performance [16, 17].

There have been several valuable studies in the database community in comparing different transactional systems [55, 13]. Harding et al. [13] conduct a comprehensive study on how different transaction protocols behave under different workloads in a distributed setting using a single framework. However, for a particular protocol, there may be many different implementations which have very different performance, especially when embracing new hardware features like RDMA.

In this paper, we conduct the first systematic study on how different choices of RDMA primitives and designs affect the performance of distributed transactions.[1] Unlike most previous research efforts which compare different overall systems, we compare different designs within

[1] Note that optimizing distributed transaction protocol is not the focus of this work.

a single execution framework. The goal is to provide a guideline on optimizing distributed transactions with RDMA, and potentially, for other RDMA-enabled systems (e.g., distributed file systems [26, 38] and graph processing systems [52, 36, 58]). In summary, this paper makes the following contributions:

A primitive-level comparison using a well-tuned RDMA execution framework (§4). We implement and tune an execution framework with all RDMA implementation techniques we know so far. We then systematically compare the performance of different primitives with existing and our newly proposed optimizations using micro-benchmarks that simulate common transactional workloads. The main results are the following (§4.2):

- One-sided primitive has better performance than two-sided with the same round trips.
- Two-sided primitive has better scalability with small payloads in large clusters.
- Two-sided primitive can be faster than one-sided when receiving ACK is done off the critical path.

A phase-by-phase evaluation of transactional execution (§5). We carefully study different primitives at different phases of transactional execution, including all optimizations proposed on both primitives, and then present their performance. More specifically, we focus on transactions with *optimistic concurrency control* (OCC)[2] for strong consistency and *primary-backup replication* for high availability. Nowadays OCC is widely used for transactions, from centralized databases [47, 49, 21] to distributed databases [11, 57, 24, 5, 18]. OCC is efficient and scalable on common workloads which stimulates many OCC-based RDMA-enabled transactions [11, 5, 18]. The protocol contains four steps: *Execution*, *Validation*, *Logging* and *Commit phase*. We show that *no single primitive always wins over the other*. To gain optimal performance for such phases, the main findings include:

- Using hybrid primitives for the execution (§5.1) and validation phases (§5.2).
- Using two-sided primitives for the commit phase (§5.3)
- Using one-sided primitives for the logging phase (§5.4).
- Using hybrid primitives and one-sided primitives for the read and validation phases of read-only transactions, respectively (§5.5).

[2]We use the shorter but more general term transactions to refer to distributed transactions executed using OCC in the rest of this paper.

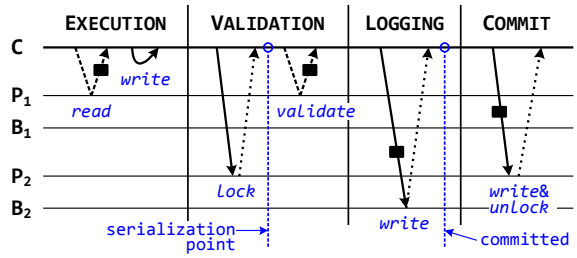


Fig. 1: *A phase-by-phase overview of transaction processing with OCC. **C**, **P**, and **B** stand for the coordinator, the primary and the backup of replicas, respectively. P_1 is read and P_2 is written. The dashed, solid, and dotted lines stand for read, write, and hardware ack operations, and rectangles stand for record data.*

An end-to-end study of existing and our new system on a single platform (§6). By further leveraging results from our phase-by-phase evaluations, we built DrTM+H, a hybrid design that optimizes every phase executed with appropriate primitives (§6.1). Evaluations using two popular OLTP workloads on a 16-node RDMA-capable cluster show that DrTM+H can perform over 7.3 and 90.4 million transactions per second for simplified TPC-C and SmallBank respectively. Further, our hybrid design does not suffer from scalability issues on an emulated 80-node connection setting (§6.2). Note that we do not make locality assumptions like previous work [18].

We finally make a comparable study on how previous systems leverage RDMA by evaluating three representative designs upon a single execution framework. To emulate previous systems, we choose the primitives and optimizations at each phase as the original design and implement them using the same codebase and transaction protocol. The experimental results show that none of them has the optimal performance (§6.3). Our hybrid design can outperform the pure two-sided design (FaSST) and the pure one-sided design (DrTM+R) by up to 2.96X and 1.89X for simplified TPC-C, respectively.

The source code of our execution framework and DrTM+H, including all benchmarks, are available at https://github.com/SJTU-IPADS/drtmh.

2 Background

2.1 RDMA and Its Primitives

RDMA (Remote Direct Memory Access) is a network feature with high speed, low latency, and low CPU overhead [10, 17]. It has generated considerable interests in applying it in modern datacenters [11, 46, 12]. RDMA is well known for its *one-sided primitive* including READ, WRITE and ATOMIC operations, which can directly access the memory of a remote machine without involving kernel and remote CPUs. Because RDMA bypasses the kernel and traditional network stack, RPC implementations over RDMA (*two-sided primitive*) can also have

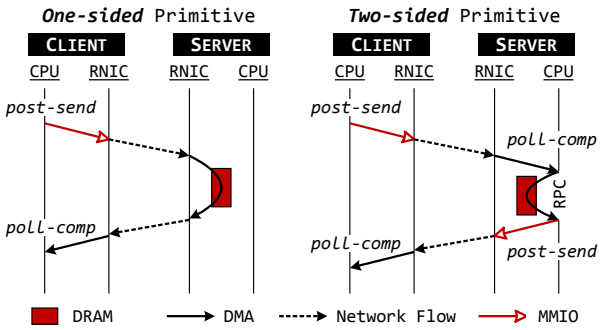


Fig. 2: *An overview of different RDMA primitives.*

orders of magnitude higher throughput than those using TCP/IP [10, 18].

Fig. 2 presents the workflow of these two primitives. No matter which primitive an application uses, the client (sender) uses a similar interface to post requests to (and poll results from) the server (receiver) via the RDMA-capable NIC (RNIC). The interface, called queue pairs (QPs), is used to communicate between the client (sender) and server (receiver). The client starts an RDMA request by posting the requests (called `Verbs`) to the sender queue, which can either be one-sided or two-sided verbs. The client can get the completion events of requests by polling a completion queue (CQ) associated with the QP. For two-sided primitives, the server polls requests from a receiver queue, calls a local RPC routine and posts results back to the sender queue.

Moreover, QPs have different transport modes which support different sets of primitives, as summarized in Table 1. The Reliable Connected (RC) mode supports all RDMA primitives, while the Unreliable Datagram (UD) mode only supports two-sided primitive (`SEND/RECV`). On the other hand, UD is connectionless so the application can use fewer UD QPs than RC QPs [18].

2.2 RDMA-enabled Distributed Transactions

There is an active line of research in using RDMA for serializable distributed transactions [11, 5, 18]. Most of such systems use variants of optimistic concurrency control (OCC) for consistency [22] and variants of primary-backup replication (PBR) for availability [23]. PBR uses fewer round trips and messages to commit one transaction than Paxos [11], which fits distributed transactions in a well-connected cluster.

Although these systems have different design choices and leverage different RDMA primitives, they use a similar transaction protocol (OCC)[3] to execute and commit serializable transactions. The operations performed in the protocol can be briefly summarized as four consecutive phases, as shown in Fig. 1. A transaction first executes by reading the records in its read set (**Execution**). Then it executes a commit protocol, which locks the records in the write set and validates the records in the read set is unchanged (**Validation**). If there is no conflicting transaction, the coordinator sends transaction's updates to each backup and waits for the accomplishment (**Logging**). Upon successful, the transaction will be committed by writing and unlocking the records at the primary node (**Commit**). Note that the execution order of the protocol is very important. For example, the transaction is considered to be committed if and only if the log replies have been received [11, 18]. Thus the commit phase must be executed after the completion of logging.

OCC can be directly used to execute read-only transactions, which is an important building block for modern applications [25]. A read-only transaction may use a two-phase protocol: the first phase reads all records (**Read**), and then the second phase validates all of them have not been changed (**Validation**).

Table 1: *Different transport modes of QP and supported operations.* ***RC****,* ***UC****, and* ***UD*** *stand for Reliable Connection, Unreliable Connection, and Unreliable Datagram, respectively.*

	SEND/RECV	WRITE	READ/ATOMIC
RC	✓	✓	✓
UC	✓	✓	✗
UD	✓	✗	✗

3 Execution Framework

To provide an apple-to-apple comparison on different primitives and transactions, we implement an execution framework for RDMA, which contains both one-sided and two-sided RDMA primitives, various prior optimizations and our newly proposed optimization.

3.1 Primitives

Symmetric model. We use a *symmetric model* in our experiments as prior work [51, 5, 11, 18]. In a symmetric model, each machine acts both a client and a server. On each machine, we register the memory with huge pages for RNIC to reduce RNIC's page translation cache misses [10].[4]

QP creation. We use a dedicated context to create QPs for each thread; otherwise, there will be false synchronizations within the driver even each thread uses its own QP. The performance impact is shown in Fig. 3(a)[5]. The root cause is that each QP uses a pre-mapped buffer to send MMIOs to post requests while the buffer may be shared. The buffer is allocated from a context according to Mellanox's driver implementation, where each context

[3] While DrTM [51] implements a two phase locking (2PL) scheme using HTM and RDMA, it provides no high availability support and a later version [5] uses a variant of OCC to provide high availability. We are not aware of other RDMA-enabled distributed transaction systems using 2PL. Hence, we focus on OCC in this paper.

[4] Currently, we use kernel's native huge page support (i.e., 2MB), which is sufficient for our current workloads.

[5] The details of experimental setup can be found in §4.1.

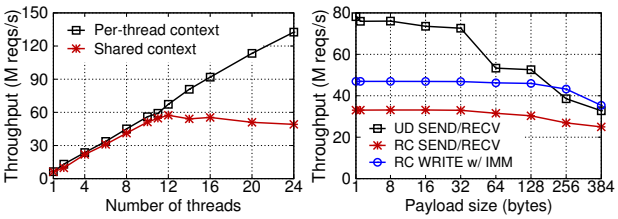


Fig. 3: *(a) The performance of RDMA* `WRITE` *using different QP creation strategies. (b) A comparison of different RDMA-enabled RPC implementations*

has limited buffers. For example, the mlx4 driver [41] uses 7 dedicated buffers and 1 shared buffer. This means that if the context is used to create more than 8 QPs, then extra QPs have to share the same buffer. Even if each thread uses one exclusive QP, the throughput of a shared context drops by up to 63% with the increase of threads. The overhead comes from synchronizations on the shared MMIO buffer.

One-sided primitive. Each thread manages n RC QPs to connect to n machines. We use standard Verbs API to post a one-sided request to the QP corresponding to the machine. RDMA `WRITE` requests with payloads less than 64 bytes are inlined to improve throughput [16]. Note that we do not simply wait until the completion of the operation (§3.2): we execute other application requests or RPC functions for better utilizing CPU and network bandwidth.

Two-sided primitive. Unlike one-sided primitive which has a simple and straightforward implementation, there are many proposed RPC implementations (two-sided primitive) atop of RDMA [10, 16, 18, 26, 46, 39]. They can be categorized into `SEND/RECV` verb based [18], RDMA `WRITE` based [10, 39, 46, 26] and hybrid one [16].

We use `SEND/RECV` verbs over UD QP as our two-sided implementation in this paper for three reasons. First, in a symmetric setting, `SEND/RECV` verbs over UD has better performance than other implementations over RDMA, especially for transaction systems [18]. This is also confirmed in our experiment (see Fig. 3(b)). Second, based on our studies of one-sided RDMA performance, *one-sided RDMA based RPC is unlikely to outperform UD based RPC* especially for small messages. The peak throughput of one-sided `WRITE` reaches 130M reqs/s when the payload size is smaller than or equal to 64 bytes (Fig. 5). For an RPC communication, two RDMA `WRITE`s are required (one for send and one for reply). Thus, the peak throughput of RPC implemented by one-sided RDMA operations is about 65M reqs/s, lower than that of the implementation based on `SEND/RECV` over UD (79M reqs/s).

Discussions. `SEND/RECV` over UD does not provide a reliable connection channel. Therefore, it may be unfair to compare it to RC based two-sided implementations which have reliability guarantees. However, since RDMA network assumes a lossless link layer, UD has much higher reliability than expected [18]. Further, packet losses can be handled by transaction's protocol [18].

3.2 Optimizations Review and Passive ACK

Many optimizations have been proposed in prior work to better leverage RDMA [10, 16, 17]. We first briefly review them here and show that when using RDMA properly, *one-sided primitive yields better performance than two-sided primitive with the same round trips.* We further propose a new optimization, *Passive ACK*, which improves RDMA primitives when the completion acknowledgement (ACK) of the request is not on the critical path of the application.

Coroutine (CO). Even the latency of RDMA operations reaches several microseconds, it is still higher than the execution time of many applications [18]. Thus, it is worth to use coroutines to further hide the network latency by sending multiple requests from different transactions in a pipelined fashion. FaSST [18] uses coroutine to improve the throughput of its RPC. FaRM [10, 11] optimizes both one-sided operations and RPCs using an event loop to schedule transactions with RDMA operations. We use a set of coroutines to execute application logic at each thread. Each coroutine yields after issuing some network requests (including both one-sided and two-sided ones), and they resume the execution until they receive the completions of one-sided requests (or the replies of two-sided RPCs). Typically, a small number of coroutines is sufficient for RDMA latency hiding (e.g., 8) [18].

Outstanding requests (OR). Even coroutine overlaps computation with I/O from different transactions, it is still important to send requests from one transaction in parallel. This further increases the utilization of RNICs and reduces the end-to-end latency of transactions, i.e., there is no need to wait for the completion of one request before issuing another one. For example, the read/write set of many OLTP transactions can be known in advance [37]. Therefore, it is possible to issue these reads and writes in parallel.

Doorbell batching (DB). There are several ways to issue multiple outstanding requests to RNIC. A common approach is to post several MMIOs corresponding to different requests. On the other hand, doorbell batching rings a doorbell to notify RNIC to fetch multiple requests by itself using DMA [17]. MMIO is costly which usually requires hundreds of cycles. Therefore, doorbell batching can reduce CPU overhead on the sender side and make

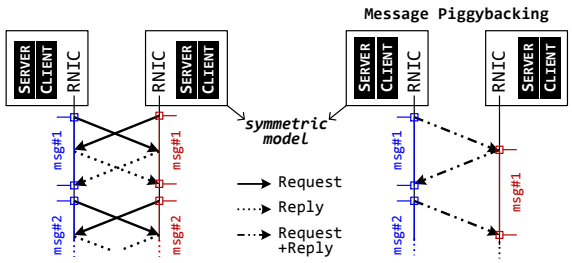


Fig. 4: *A sample of passive ACK for two-sided primitive.*

a better usage of PCIe bandwidth, since it only requires one MMIO per batch to ring the doorbell.

One restriction of doorbell batching is that only requests from one QP can be fetched by the RNIC in a batched way. This means that different one-sided requests cannot be batched together if they are not sent to the same machine. Due to this limitation, doorbell batching is usually applied to two-sided implementation based on UD QP [18].

Passive ACK (PA). The performance can be further improved if the completion of requests (ACK) is done off the critical path of transactional execution. We achieve this by acknowledging the request passively.

For one-sided primitive, the request is marked as unsignaled, and then the completion of the request is confirmed passively after a successful polling of one subsequent signaled request. This avoids consuming RNIC's bandwidth.[6] For two-sided primitive, the optimization has the potential to double the throughput in a symmetric model by piggybacking the reply messages with the request messages. As shown in Fig. 4, passive ACK can save half of the messages (replies).

It should be noted that not all of the completions can be acknowledged passively. For example, one-sided `READ` requires a completion event; otherwise, the application does not know whether the read is successful or not. Fortunately, in transactional execution, a transaction is considered to be committed when the log has been successfully written to all backups (see Fig. 1). Hence the write-back request at the commit phase can be acknowledged passively.

4 A Primitive-level Performance Analysis

In this section, we first present our execution framework with both one-sided and two-sided primitive of RDMA. We then present the basic performance of different RDMA primitives, including raw RDMA performance and the performance of micro-benchmarks. The micro-benchmarks simulate common transactional workloads. These experimental results serve as the guideline for using the appropriate primitives for transactions.

[6] Verbs from the same send queue are processed in a FIFO manner [2].

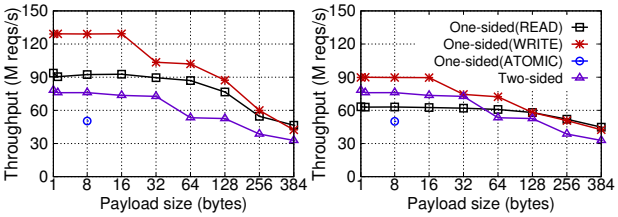


Fig. 5: *A comparison of one-sided and two-sided primitives using (a) a 16-node cluster and (b) an emulated 80-node connection setting. RDMA `ATOMIC` only supports the 8-byte payload.*

4.1 Setup

Testbed. Unless otherwise specified, we use a local rack-scale RDMA-capable cluster with 16 machines for all experiments. Each machine is equipped with two 12-core Intel Xeon E5-2650 v4 processors, 128GB of RAM, and two ConnectX-4 MCX455A 100Gbps Infiniband NIC via PCIe 3.0 x16 connected to a Mellanox SB7890 100Gbps InfiniBand Switch.

Execution. We run 24 worker threads (same as the number of available cores per machine) on each machine in our experiments. Each worker thread runs an event loop to execute transactions, handles RPC requests, and polls RDMA events. The events of RDMA including the completion of one-sided RDMA requests and the reception of RPC requests/replies. We follow FaSST [18] by using coroutine from Boost C++ library to manage context switches between clients when issuing network requests. Boost coroutine is efficient in our experiments, which has very low overhead for context switch (about 20 ns).

4.2 Primitive-level Performance Analysis

RDMA raw performance. Prior work has shown that two-sided primitives have better performance and scalability than one-sided ones [18]. This conclusion is drawn from an old generation of RNIC (ConnectX-3). Further, they only show the poor scalability of one-sided primitive using small payloads (less than 32 bytes). We extend their evaluation [18] on raw RDMA performance to show that: one-sided primitives have better performance than two-sided ones using 16 nodes, as shown in Fig. 5(a). More importantly, the scale of the cluster only affects *one-sided primitives with small payloads*. For example, with our emulated 80-node connection setting, one-sided primitives still outperform two-sided ones when data payloads are larger than 64 bytes.

Emulating massive RDMA connections. On our 16-node cluster, we create 5 RC QPs to connect to each machine at each worker. The number of QPs (5x16 QPs per thread) is sufficient to run in an 80-node cluster. We choose the QPs randomly to post upon issuing a request. Note that the total number of QPs (960 per NIC) has exceeded the total number of QPs that can be cached at RNIC.

Primitive evaluation. Fig. 5(a) presents the evaluation results of the primitive analysis. For read operations, one-sided primitives (`READ`) outperform two-sided ones by up to 1.6X when payload size is below 64 bytes, and by up to 1.37X for larger payloads. For write operations, one-sided primitives (`WRITE`) outperform reads on small payloads but get a similar trend on large payloads (from 1.03X to 1.35X). Note that we do not incur memory copy overhead for two-sided primitives, as done in prior work [18], since adding such overhead will affect the performance of two-sided ones, especially for large messages.

Fig. 5(b) further presents the results on an emulated 80-node connection setting. The performance of one-sided `READ` becomes slow-growing with the decrease of payloads from 128 bytes. This is because RNIC experiences QP cache misses at this time.[7] However, one-sided `READ`s can still outperform two-sided primitives when payloads are larger than 64 bytes. Because the cost of data transfer instead of QP cache misses dominates the performance for larger payloads.

A final takeaway is that, although one-sided `ATOMIC` is relatively slow [17], it can still achieve 48M reqs/s on each machine, which is much higher than the requirements of many workloads (e.g., `TPC-C`). Therefore, the performance will not be the main obstacle to leverage one-sided atomic primitives in transactional execution (e.g., distributed spinlock). We evaluate this approach in the transactional workload (§5.2).

Micro-benchmarks. Better performance in raw throughput does not always mean better performance in real applications. We use two micro-benchmarks to compare how different primitive performs under common transactional workloads, and how previous optimizations affect the performance (Fig. 6).

Workloads. We use a workload with multi-object reads and writes to compare the performance of different RDMA primitives. This workload simulates common operations in transactional workloads: at the execution phase, transaction reads multiple records from remote servers; at the commit phase, transaction writes multiple updates back to remote servers. Note that the workloads use fixed-length 64-byte payloads, and issue 10 operations.

Effects of optimizations. We first show how existing optimizations improve the performance of each primitive from Fig. 6. Coroutine, outstanding requests and doorbell can be applied to both workloads.

Coroutine hides the latency and improves the performance of one-sided and two-sided by 7X and 6.46X, respectively. Adding outstanding requests by posting more requests per batch further improve the throughput due to better uses of RNIC's processing capability.

[7] We use PCIe counters to measure QP cache misses, similar to pmu-tools (https://github.com/andikleen/pmu-tools).

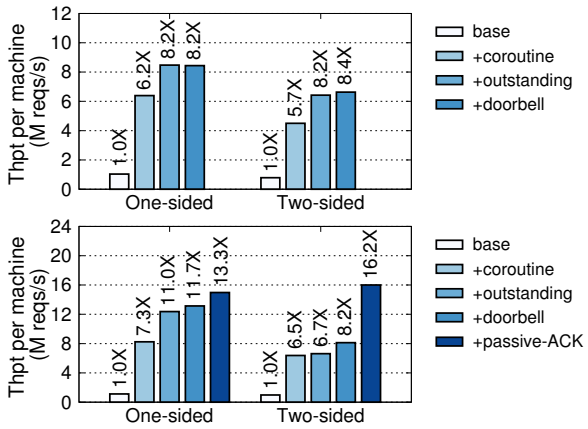


Fig. 6: *A comparison of one-sided and two-sided primitives for multiple-object (a) reads and (b) writes with 64-byte payloads.*

Doorbell batching does not always improve the performance of one-sided primitive, but it constantly improves the throughput of two-sided ones. This is because doorbell batching can only apply to a single QP, which is suitable for UD-based two-sided implementation. On the contrary, one-sided requests are sent through multiple RC QPs, which reduces the chances of using doorbell batching. Further using doorbell batching requires bookkeeping the status of posted requests, which adds additional overhead.

Offloading when completion is required. By enabling passive acknowledgement (PA), the performance of one-sided `WRITE` is further improved by 1.13X, while that of two-sided primitive is nearly doubled (1.96X) due to the reduction of half of the messages (for reply). This makes the only case where two-sided outperforms one-sided. Otherwise, one-sided primitive always has better performance than two-sided ones. This is consistent with the results in Fig. 5. For example, multiple `READ`s can achieve peak throughput about 8.43M, which is close to the raw performance of one-sided `READ` (about 86.9M per machine).

5 A Phase-by-phase Performance Analysis

In this section, we study the performance of transactions phase-by-phase with different RDMA primitives. Table 2 summaries whether we apply the optimization discussed in §3.2 at different phases of transactional execution. Below is some highlights of our phase-by-phase analysis:

- One-sided primitive is faster when the number of round trips is the same and the completion acknowledgement of requests are required (§5.1,5.2,5.4,5.5).
- It is always worth checking and filling the lookup

cache for one-sided primitive, even using two-sided primitive (§5.1).

- One-sided primitive is faster, even using more network round trips, for CPU-intensive workloads (§5.1).
- Two-sided primitive with passive ACK has comparable or better performance than one-sided (§5.3).

Benchmarks. We use two popular OLTP benchmarks, TPC-C [44] and SmallBank [42], to measure the performance[8] of every phase with different primitives, since they represent *CPU-intensive* and *network-intensive* workloads respectively. We use a partitioned data store where data is sharded by rows and then distributed to all machines. We enable 3-way logging and replication to achieve high availability, namely each primary partition has two backup replicas.

TPC-C simulates an order processing application. We scale the database by deploying 384 warehouses to 16 machines. We use this benchmark as a CPU-intensive workload. TPC-C is known for good locality: only around 10% of transactions access remote records. To avoid the impact of local transactions, which our work does not focus on, we only run `new-order` transaction of TPC-C and make transactions always distributed, which is a major type of transaction (45%) and representative in TPC-C.[9]

SmallBank simulates a simple banking application. Each transaction performs simple reads and writes operations on account data, such as transferring money between different users. We use this benchmark as a network-intensive workload because transaction only contains simple arithmetic operations on few records. We do not assume locality as previous work [18], which means that all transactions use network operations to execute and commit transactions. To scale the benchmark, we deploy 100,000 accounts per thread, while 4% of records are accessed by 90% of transactions.

5.1 Execution (E)

Overview. The transaction coordinator fetches the records a transaction reads in the *execution* phase. This requires traversing the index structure and fetching the record. We can simply send an RPC to remote server to fetch the record, which only requires one round-trip communication. On the other hand, we can also leverage one-sided RDMA READs to traverse the data structure and read the record. This typically requires multiple round trips but saves remote CPUs. Prior work has proposed two types of optimizations to reduce the number of round trips required by one-sided primitives [29, 10, 51, 30].

[8] We scale up the concurrent requests handled by the server to achieve the peak throughput.

[9] For brevity, we refer to our simplified TPC-C benchmark as TPC-C/no.

Table 2: *A summary of optimizations on RDMA primitives at different phases (§3.2).* ***OR****,* ***DB****,* ***CO*** *and* ***PA*** *stand for outstanding request, doorbell batching, coroutine, and passive ACK.* ***RW*** *and* ***RO*** *stand for read-write and read-only transactions.* `I` *and* `II` *stand for one-sided and two-sided primitives.*

		OR		DB		CO		PA	
		I	II	I	II	I	II	I	II
RW	E	✗	✗	✗	✓	✓	✓	✗	✗
	V	✓	✓	✓	✓	✓	✓	✗	✗
	L	✓	✓	✗	✓	✓	✓	✗	✗
	C	✓	✓	✓	✓	✓	✓	✓	✓
RO	R	✗	✗	✗	✓	✓	✓	✗	✗
	V	✓	✓	✗	✓	✓	✓	✗	✗

RDMA-friendly key-value store. Many hash-based data structures can be optimized to reduce the number of RDMA operations for traversing the remote server to find the given key, these include cuckoo hashing [29], hopscotch hashing [10], and cluster hashing [51]. We adopt DrTM-KV [51], a state-of-the-art RDMA-friendly key-value store in all experiments.

RDMA-friendly index cache. The ideal case for one-sided primitive is to use one one-sided READ to get the record back. DrTM [51] introduces a location-based cache to eliminate the lookup cost (one RDMA READ) in the common case. FaRM [11] and Cell [30] use a similar design for caching the internal nodes of B-tree. In our experiment, we maintain a 300MB index cache on each machine, which will be used and filled in the execution phase. Note that the index cache is quite effective since a relatively small cache is usually enough for skewed OLTP workloads [15, 8, 3, 33, 20], such as SmallBank [42], TATP [32], and YCSB [6].

Evaluation. Fig. 7 compares the performance of using one-sided and two-sided primitives for the execution phase on TPC-C/no and SmallBank, respectively. Two-sided uses one RPC to fetch the record. One-sided fetches records with at least two one-sided READs (one for index and one for payload). One-sided/Cache always fetches the indexes from the local index cache and then get the record from a remote server using a single one-sided READ. This presents an ideal case for the performance of the execution phase using one-sided primitives.

TPC-C/no: One-sided/Cache outperforms Two-sided by up to 1.45X in throughput (from 1.26X), and the median latency is only around 69% of Two-sided (from 89%). The benefits mainly come from the better performance of one-sided READs. Two-sided outperforms One-sided (no cache) by up to 1.28X in throughput. Without caching, the coordinator requires an average of double round trips (one for lookup and another for read) to fetch one record.

Interestingly, when increasing the number of corou-

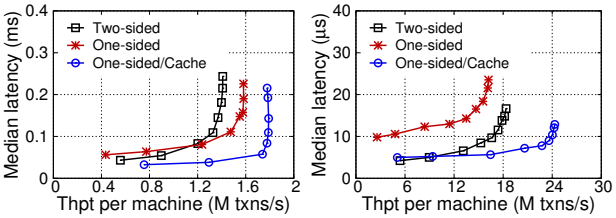


Fig. 7: *The performance of (a) TPC-C/no and (b) SmallBank with different implementations of Execution phase.*

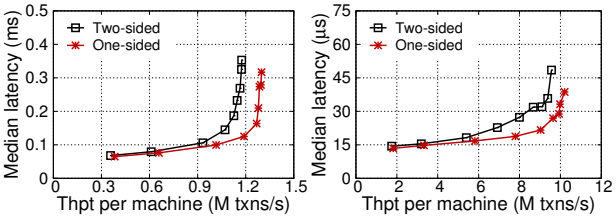


Fig. 8: *The performance of (a) TPC-C/no and (b) SmallBank with different implementations of locking in Validation phase.*

tines, the peak throughput of One-sided (no cache) outperforms that of Two-sided (about 13%). The median latency is also slightly better when using more than 10 coroutines. The performance gain comes from lower CPU utilization on each machine. This confirms the benefits of using one-sided primitives when remote servers are busy [29, 30]. The adaptive caching scheme in prior work [29, 30] can be used to get better performance by balancing CPU and network.

SmallBank: Not surprisingly, One-sided/Cache still outperforms Two-sided by up to 1.36X in throughput due to the better performance and CPU utilization of one-sided READ. However, compared to One-sided (no cache), the speedup of peak throughput for Two-sided reaches up to 2.01X (from 1.13X). This is due to two reasons. First, without location-based cache, one-sided uses more round trips to finish the execution phase. Further, the performance of Smallbank is bottlenecked by network bandwidth since it is a network-intensive workload.

Summary. If one round-trip RDMA READ can retrieve one record using the index cache, one-sided primitive is always a better choice than two-sided one. Otherwise, two-sided primitive should be used when servers are not overloaded. Hence, a *hybrid* scheme should be used in the execution phase. Specifically, we should *always enable the index cache and look from it* before choosing either one-sided primitive (on cache hit) or two-sided primitive (on cache miss). We should also *always refill the cache even if two-sided primitive is chosen upon a miss*.

5.2 Validation (V)

Overview. To ensure serializability, OCC atomically checks the read/write sets of the transaction in the validation phase. The coordinator first *locks* all records in the transaction's write set and then *validates* all records in the read/write set to ensure that they have not been changed after the execution phase.

Lock. RDMA provides one-sided *atomic compare and swap* operations (ATOMIC), which can be used to implement distributed spinlock [51, 5]. Although ATOMIC is slower than other two-sided primitives, on recent generation of RNIC (e.g., ConnectX-4), ATOMIC can achieve 48M reqs/s, which is enough for many OLTP workloads (e.g., TPC-C). More importantly, the throughput of two-sided primitive (76M) was evaluated with an empty RPC workload. When locking the record in the RPC routine, the impact of CPU efficiency may change the relative performance of one-sided and two-side primitives. This is especially the case for the symmetric model adopted by transaction systems [51, 5, 11, 18, 56], when the servers are busy processing transactions.

Validate. Different from the execution phase, a single RDMA READ is enough to retrieve the current version of the record for validation, thanks to caching the index in the execution phase of the transaction. Therefore, one-sided primitive is always a better choice for read-only records compared to two-sided one, according to the results in Fig. 5 and Fig. 7.

Optimization. OCC demands the validation should start exactly after locking all records [47, 11]. This takes two round trips for every read-write record in the validation phase. Fortunately, the locked record can be validated immediately since it can not be changed again. Therefore, each read-write record can be handled by both one-sided and two-sided primitives in one round trip. For one-sided, the RDMA READ request will be posted just after the RDMA CAS request in a doorbelled way to the same send queue of target QP, since they are processed in a FIFO manner. Further, with passive ACK, the CAS request can be made unsignaled (§3.2). For two-sided, the RPC routine will first lock the record and then read its version. On commodity x86 processors, compiler fences are sufficient to ensure the required ordering.

Restriction of RDMA atomicity. Currently, the key challenge for using one-sided primitive (RDMA ATOMIC) for distributed locking is that ATOMIC cannot correctly work with CPU's atomic operations (e.g., CAS). To remedy this, local atomic operations must also use RNIC's atomic operations [51], which will slow down the validation phase of local transactions. Leveraging advanced hardware features, like hardware transactional memory (HTM), can overcome this issue [51].

Evaluation. Fig. 8 compares the performance of using one-sided and two-sided primitives for the validation phase on TPC-C/no and SmallBank, respectively.

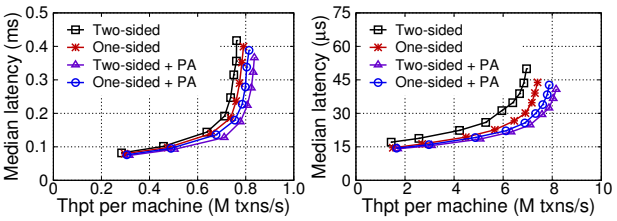


Fig. 9: *The performance of (a) TPC-C/no and (b) SmallBank with different implementations of Commit phase.*

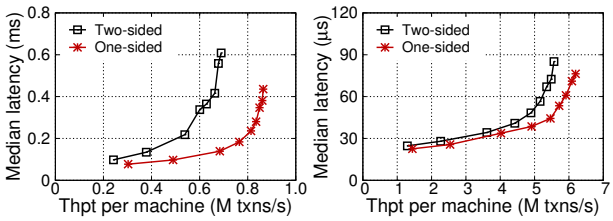


Fig. 10: *The performance of (a) TPC-C/no and (b) SmallBank with different implementations of Logging phase.*

Since the read/write sets are the same in TPC-C/no and SmallBank, one-sided will send one ATOMIC and one READ sequentially to lock the record and retrieve the current version in one round trip. We can see in Fig 8 that for both workloads, one-sided primitive (ATOMIC) is faster, even it has lower peak throughput.

Summary. Although RDMA ATOMIC is slower than other RDMA network primitives, it may not be the bottleneck for many applications and can further improve the performance of many workloads. If the atomicity between RNIC and CPU will not cause a performance issue, One-sided RDMA ATOMIC is a better choice to implement distributed locking due to high CPU efficiency. Otherwise, two-sided primitive is preferred in this phase since local CASs are much faster than RNIC's CASs.

5.3 Commit (C)

Overview. In the commit phase, the coordinator first writes the updates of the transaction back and then releases the locks. One-sided WRITE can be used to implement the commit operation with two requests, one to write updates back and one to release the locks (i.e., zeroing the lock state of the record).

Similar to the validation phase, two one-sided WRITEs (one to write the update back and one to release the lock) will be posted sequentially to the same QP in a doorbelled way, which preserves the required ordering (release after write-back). Therefore, the commit phase can be handled by both one-sided and two-sided primitives in one round trip.

Optimization with passive ACK. Since the transaction is considered to be committed after the completion of logging, the completion of the commit message can be acknowledged passively by piggybacking with other messages. Thus we enable passive ACK optimization to both one-sided and two-sided primitives in the commit phase.

Evaluation. Fig. 9 presents the performance of TPC-C/no and SmallBank using different commit approaches. Note that we use two-sided as the validation implementation in this experiment. This is because one-sided ATOMICs cannot work correctly with the commit phase with two-sided primitive due to the atomicity issue with our current RNIC.

For both workloads, without passive ACK, one-sided WRITEs are faster due to better CPU utilization at the receiver's side. With passive ACK, two-sided is faster. This is because, although two-sided primitive costs more CPU at the receiver side, it can save CPU at sender side due to doorbell batching [17] (see Table 2). One-sided primitive requires multiple MMIOs to commit multiple records, while two-sided primitive can chain these requests by using one doorbell. Passive ACK can further save the cost of two-sided primitives when sending the replies back. These results match up with the results observed in our primitive-level performance analysis (§4).

Summary. To commit transactions, two-sided primitive with passive ACK is the better choice.

5.4 Logging (L)

Overview. In the logging phase, the coordinator writes transaction logs with all updates to all backups. After receiving the completion acknowledgements from all backups, the transaction commits. The coordinator will notify backups to reclaim the space of logs lazily by updating records in-place.

One-sided primitive. To enable logging with one-sided RDMA WRITE, each machine maintains a set of ring-buffers for remote servers to log. The integrity of the log is enforced by setting the payload size at the begin and end of the message, similar to previous work [10]. Note that since we use RC (Reliable Connection) QP to post one-sided WRITEs, the logging is considered success after polling the ACK from the RNIC. We use two-sided primitive to reclaim the log since it must involve remote CPUs [11]. Since log reclaiming is not on the critical path of transactional execution, this request can be marked as unsignaled and the claiming can be done in the background.

Two-sided primitive. Logging with two-sided primitive is relatively simple. The RPC routine copies the log content to a local buffer after receiving the log request, and then it sends a reply to the sender. The log reclaiming can also be executed in the background.

Evaluation. Fig. 10 presents the performance of TPC-C/no and SmallBank using different logging approaches. For both of them, one-sided logging always

Table 3: *A summary of execution time (cycles) and payload size (bytes) in different phases for TPC-C and SmallBank.*

	TPC-C		SmallBank	
	Time	Payload	Time	Payload
Execution	342	68	678	71
Validation	454	157	185	105
Logging	363	1006	134	149
Commit	108	34	87	20

has higher throughput and lower latency than its two-sided counterpart, thanks to offloading write operations to one-sided primitives. Using one-sided logging increases the throughput of TPC-C/no and SmallBank by up to 1.29X (from 1.24X) and 1.12X (from 1.10X), respectively. One-sided logging has more improvements in peak throughput in TPC-C/no since the payload size of logs in TPC-C is much larger than that of SmallBank (1,006B vs. 149B), as shown in Table 3.

Summary. Since the logging phase can be offloaded using one-sided RDMA WRITEs with one round trip, one-sided primitive is always preferred to write logs.

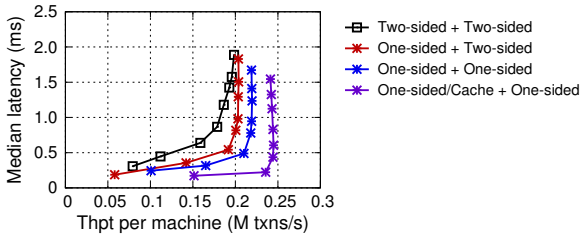


Fig. 11: *The performance of customer-position in TPC-E with different implementations of the read-only transaction (Read and Validation phases).*

5.5 Read-only Transaction (R+V)

Overview. We use a simplified two-phase protocol to run read-only transactions as prior work [25]. The first phase reads all records like the execution phase, and the second phase validates that the versions of all records have not been changed, which is similar to the operations in the validation phase for the records in read set. For single-key read-only transactions, the validation phase can be ignored. These transactions are popular in many OLTP workloads (e.g., TATP [32]), as reported by prior work [11, 18].

Evaluation. With a proper sharding, there is no distributed read-only transaction in TPC-C, which needs remote data accesses. Further, there is only one single-key read-only transaction in Smallbank (i.e., Balance), which does not require the second phase (validation) [11, 18]. Therefore, we use the customer-position transaction in TPC-E [43] to evaluate the performance of distributed read-only transactions.

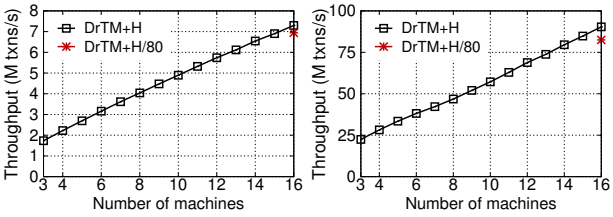


Fig. 12: *The performance of DrTM+H with the increase of machines for (a) TPC-C/no and (b) SmallBank.*

TPC-E. is designed to be a more realistic OLTP benchmark, which simulates the workload of a brokerage firm. One of well-known characteristics is the high proportion of read-only transactions, reaching more than 79%. The customer-position transaction is read-only and has the highest execution ratio. It simulates the process of retrieving the customer's profile and summarizing their overall standing based on current market values for all assets. The assets prices are fetched in a distributed way.

Fig. 11 compares different choices of primitives for distributed read-only transactions. As expected, by offloading read operations to RNICs and bypassing remote CPUs, using one-sided primitives for both the read and validation phases can gain the best performance in both throughput and latency. One-sided outperforms Two-sided by about 10% in peak throughput (0.19 vs. 0.21), and the median latency is around 80% of Two-sided. Enabling the index cache (One-sided/Cache) in the read phase will further improve the peak throughput by close to 20% (0.25 vs. 0.21) and reduce the median latency more than 20%.

Summary. The hybrid scheme used in the execution phase (see §5.1) is also suitable to the first phase, and one-sided READ is always a better choice for the second phase (see §5.2). For single-key read-only transactions, a single one-sided READ is usually efficient.

6 Fast Transactions using Hybrid Schemes

In this section, we conclude our studies of using RDMA primitives for transactions by showing how to improve the performance of prior designs by choosing appropriate primitives and techniques at different phases of transactional execution. This leads to DrTM+H, an efficient distributed transaction system using hybrid schemes.

6.1 Design of DrTM+H

DrTM+H optimizes different phases of the transaction by choosing the right primitives guided by our previous studies (§4 and §5). DrTM+H supports serializable transaction with log replication for high availability. Currently, we have not implemented the reconfiguration and recovery, which is necessary to achieve high availability. Yet, since our replication protocol is exactly the same as the one

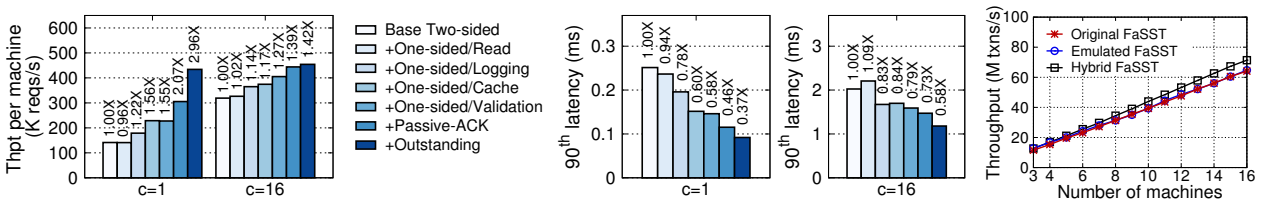


Fig. 13: *The contribution of optimizations to (a) throughput and (b,c) latency to* `TPC-C/no` *for* `DrTM+H` *using 1 and 16 coroutines, respectively. Optimizations are cumulative from left to right. (d) A performance comparison between original and emulated FaSST.*

used in FaRM [11], `DrTM+H` can use its method to recover from failure.

Execution. `DrTM+H` uses a hybrid design of one-sided `READ`s with caching and two-sided RPC. If the record's address has been cached locally, one RDMA `READ` is sufficient to fetch the record. Otherwise, `DrTM+H` uses RPC to fetch the record and its address.

Validation. `DrTM+H` uses one-sided `ATOMIC` for validation if there is no atomic issue (e.g., Network accesses do not conflict with local ones). Otherwise two-sided is preferred since using RDMA atomic operations will slow down local operations [51].

Logging. `DrTM+H` always uses one-sided `WRITE`s to replicate transaction logs to all backups and uses two-sided primitive to lazily reclaim logs on backups.

Commit. `DrTM+H` uses one-sided `WRITE`s to commit if one-sided `ATOMIC` is used in the validation phase. Otherwise `DrTM+H` uses two-sided RPC. `DrTM+H` always uses passive ACK optimization since the completion of commit message is not on the critical path of transactional execution.

Using outstanding request with speculative execution. In §5.1, we disable the outstanding request optimization at the execution phase to avoid requiring advance knowledge of read/write set. However, this usually means that transaction must fetch records one-by-one, which increases the latency of a single transaction.[10] We found that even the record has not been fetched to local, the transaction can still speculatively execute until the involved value is really used. This can greatly reduce the lifespan of a transaction. For example, the remote records required by `new-order` transaction in `TPC-C` are independent. Thus `DrTM+H` uses speculative execution to fetch these records in parallel.

6.2 Performance Evaluation

Fig. 12 presents the throughput and scalability of `DrTM+H` using `TPC-C/no` and `SmallBank`. To show that `DrTM+H`'s usage of one-sided primitive has good scalability on a larger-scale cluster, we use the QP setting which is enough to run on an 80-node cluster (`DrTM+H`-80). Each thread uses 80 QPs (16x5) to connect to 16 nodes and chooses the usage of QP in a round-robin way.

[10]We still send multiple requests in parallel for different transactions using coroutines.

Table 4: *A review of the existing RDMA-enabled transaction systems.* `I` *and* `II` *stand for one-sided and two-sided primitives.*

	RW-TX				RO-TX	
	E	V	L	C	R	V
FaRM	I	II+I	I	II	I	I
DrTM+R	I	I+I	I	I+I	I	I
FaSST	II	II	II	II	II	II
DrTM+H	I/II	I/II	I	I/II	I/II	I

Performance and scalability. `DrTM+H` scales linearly with the increasing of machines. The throughput of `TPC-C/no` and `SmallBank` decrease 5% and 9% on the emulated 80-node connection setting, respectively. `SmallBank` is more sensitive to the number of QPs since its payload size is much smaller than that of `TPC-C/no`. However, `SmallBank` is still 1.3X higher than a pure two-sided solution in throughput, with a significant decrease in the tail latency. The 50^{th} (median), 90^{th}, and 99^{th} latency are reduced by 22%, 39%, and 49%, respectively.

Factor analysis. To investigate the contribution of the primitive choices in `DrTM+H`, we conduct a factor analysis in Fig. 13. Due to space limits, we only report the experimental results of `TPC-C/no`; `SmallBank` is similar. First, we observe that using one-sided primitives can significantly improve the throughput and latency when servers are underloaded (1 coroutine). This is because one-sided primitive has lower CPU utilization and lower latency compared to two-sided one. Second, by increasing coroutines, the two-sided implementation has close throughput with one-sided one. However, a hybrid scheme in `DrTM+H` improves both median and tail latency. Finally, when leveraging RDMA, the number of round trips has more impacts on latency but not throughput, especially for CPU-intensive workloads (e.g., `TPC-C`). When using 16 coroutines, the throughput increases even using more network round trips (adding one-sided `READ`s). This is because coroutines hide most of waiting for request's completion while one-sided primitive has lower CPU utilization.

6.3 Comparison Against Prior Designs

There have been several designs to optimize transactional execution using RDMA. To understand the effects of RDMA primitive decisions, we implemented and evaluated emulated versions of FaRM [11], DrTM+R [5] and FaSST [18].[11] We adopted the same codebase and transaction protocol (OCC) of `DrTM+H`, but choosing the RDMA primitives and techniques at different phases of transactional execution as the originals. Table 4 summarizes the primitives used in the three systems and compares the performance of emulated versions of them with `DrTM+H`. Note that all existing optimizations on RDMA primitives are enabled, including coroutine, outstanding requests, and doorbell batching.

Emulating FaRM. FaRM [11] is designed to run transactions atop of a global memory space over RDMA networking. FaRM uses one-sided `READ` at the execution/logging phase and one-sided `WRITE` at the logging phase, as well as a hybrid choice at the validation phase. Moreover, FaRM adopts an RDMA-friendly memory store (FaRM-KV) proposed in their prior work [10]. Our emulated store (DrTM-KV) has been shown to have a comparable performance even without the location cache [51]. Further, our two-sided RPC implementation has also better performance than the implementation in FaRM [18] (see `RC WRITE w/ IMM` in Fig 3(b)). Hence, we believe our emulated version has similar or even better performance compared to the vanilla FaRM.

Emulating DrTM+R. DrTM+R [5] offloads all network operations to one-sided RDMA primitives for CPU efficiency, including using one-sided `ATOMIC` for locking remote records in the validation phase. Further, DrTM+R exploits hardware transactional memory (HTM) [14] to handle local transactions, but does not leverage coroutines to obtain higher throughput. To focus on comparing different choices of RDMA primitives, our emulated version disables HTM (similar to the implementation of DrTM-OCC [4]) but enables coroutine optimization.

Emulating FaSST. FaSST [18] proposes a well-optimized RPC implementation (see `UD SEND/RECV` in Fig 3(b)) based on two-sided primitives for running transactions. Since our framework provides a similar UD-based RPC implementation, it is straightforward to emulate FaSST by using two-sided primitives at all phases of transactional execution. Since FaSST uses a simplified OCC protocol [18] by moving lock operations from the validation phase to the execution phase, we use FaSST-OCC to name the pure two-sided implementation on our platform with OCC protocol, to avoid confusion.

[11] FaRM is not open-sourced, `DrTM+R` depends on hardware transactional memory, and FaSST uses a simplified OCC and protocol.

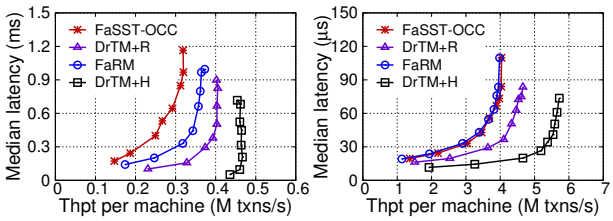


Fig. 14: *An end-to-end comparison of different designs for (a)* `TPC-C/no` *and (b)* `SmallBank`.

Calibrating performance with FaSST. Since the source code of FaSST is available and does not depend on specialized hardwares, we compare the performance of the original version and our emulated version using the `SmallBank` benchmark with 24 threads and 1 NIC per machine.[12] Note that we also implement their specific OCC protocol which can reduce one network round trip with advance knowledge of transaction's read/write set.As shown in Fig. 13(d), emulating FaSST on our platform can achieve comparable performance. Further, under our guideline for using the appropriate primitives, a hybrid design can also improve the implementation of FaSST's protocol by using one-sided primitives in the validation phase and the logging phase. As shown in Fig. 13(d), the hybrid choice (Hybrid FaSST) outperforms the original FaSST by up to 11% in throughput.

Evaluation. Compared to other prior designs, `DrTM+H` always embraces the best performance in terms of latency and throughput. Fig. 14 presents our results. `DrTM+H` has the best throughput than previous designs with the right choice of RDMA primitives and a set of optimizations to better leverage the chosen primitive. On `TPC-C/no`, `DrTM+H`'s throughput is up to 2.96X of FaSST (from 1.41X), up to 1.89X of DrTM+R (from 1.12X) and up to 2.50X of FaRM (from 1.21X). When using 16 coroutines, the median latency is reduced by 33%, 23% and 34%, respectively. We broke down the performance improvements in §6.2. FaRM optimizes baseline two-sided (FaSST) by using one-sided operation for logging and execution. DrTM+R further adds location cache and use one-sided for validation and commit. In `TPC-C/no`, FaRM and DrTM+R outperforms FaSST due to better leveraging one-sided primitives for CPU-intensive workloads. DrTM+R outperforms FaRM due to the usage of location cache at the execution phase and the usage of atomics at the validation phase. FaSST has a comparable performance to FaRM for `SmallBank` since two-sided primitive is faster at the execution phase.

Latency breakdown. To study the performance influence of choosing RDMA primitives, we further show the latency breakdown in each phase for different designs

[12] The original FaSST does not support the `TPC-C` benchmark, and we utilized the configuration as suggested by the authors of FaSST.

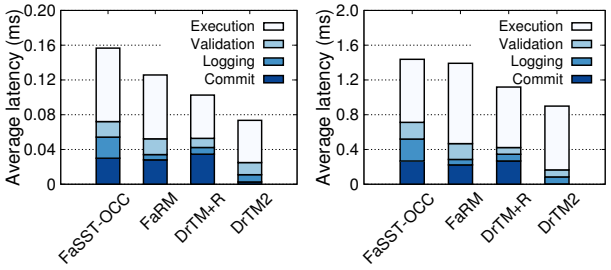


Fig. 15: *The latency breakdown of TPC-C/no using (a) 1 coroutine and (b) 16 coroutines.*

in Fig. 15. By leveraging one-sided READs, the latency of the execution phase is reduced by 13% and 41% in FaRM and DrTM+R respectively, when using one coroutine. However, the increase of coroutines can narrow the performance gap by hiding the latency of network operations. FaSST can outperform FaRM by 22% when using 16 coroutines, since FaRM requires more network round trips to read remote data. To remedy this, DrTM+R enables the location-based cache [51] for one-sided operations and achieves the lowest latency (less than 0.7ms). In the validation phase, DrTM+R has the lower latency by offloading lock operations to RDMA NICs. Using one-sided WRITEs, the latency of the logging phase in DrTM+R and FaRM is reduced by about 69% and 75% respectively, compared to using two-sided primitives (FaSST). Finally, DrTM+H can always choose appropriate RDMA primitives to embrace the latency reduction at each phase. Note that DrTM+H has the lowest latency at the commit phase due to enabling Passive ACK optimization (§3.2), such that receiving the acknowledgement of commit messages is done off the critical path.

Table 5: *Different RDMA NICs used in the experiments. N denotes the number of nodes. C/N denotes the number of RNICs per node. P/C denotes the number of ports per RNIC. Each machine is the same as in §4.1.*

Name	N	C/N	P/C	RNIC
CX3	5	2	1	40Gbps ConnectX-3 InfiniBand
RoCE	2	2	1	100Gbps ConnectX-4 RoCE
CX5	2	1	1	100Gbps ConnectX-5 InfiniBand

7 Other RDMA NICs

The design choice of using specific RDMA primitive is guided by our primitive analysis in §4. The analysis itself depends on the performance of RNIC for different primitives. So our results depend on specific RNIC hardware characteristics.

In this section, we provide experiments using different RDMA platforms to show how our results applied to other settings. The experiment settings are summarized in Table 5. In summary, if RNIC can provide better performance for one-sided primitive using the same round

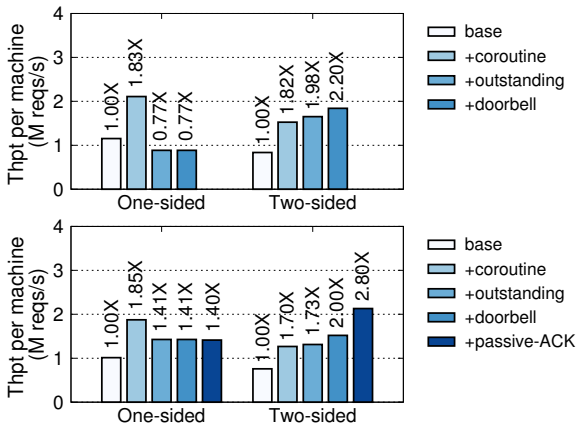


Fig. 16: *A comparison of one-sided and two-sided primitives using ConnectX-3 RNICs for multiple-object (a) reads and (b) writes with 256-byte payloads.*

trip, our results generally hold. We have observed faster one-sided primitive in recent generations of RNICs, like ConnectX-4 (CX4) and ConnectX-5 (CX5). On the other hand, if one-sided primitive cannot provide better performance, which is the case in old generations of RNIC, like ConnectX-3 (CX3), two-sided primitive shall be used.

ConnectX-3 (CX3). CX3 is an old generation of RNIC released in 2011. It is well-known for its poor one-sided performance [17, 18]. Using our micro-benchmarks, as shown in Fig. 16, one-sided primitive cannot provide better performance than two-sided primitive even using the same number of network round trips. Further, one-sided ATOMICS blocks RNIC operations [17] in CX3, resulting in relatively poor performance.

The poor performance of one-sided primitive in CX3 makes two-sided primitive a better design choice. This is because UD based two-sided primitive is less affected by hardware restrictions [16, 17]. Fig. 17 shows the end-to-end comparison of prior designs we reviewed in §6 using CX3. We can see that FaSST achieves the best performance due to better performance of two-sided primitives at each phase. FaRM uses slower one-sided primitive with more round-trip at the execution phase (for traversing the index). DrTM+R achieves the slowest performance because its performance is bottlenecked by the one-sided ATOMICS. Yet, DrTM+H can still choose the right primitive based on the evaluation results and achieve the best results.

RDMA over Converged Ethernet (RoCE). RoCE is a network protocol which allows RDMA to run atop of an Ethernet network. It uses Ethernet as the link layer compared to the Infiniband network. Since RoCE only uses a different link layer, it usually has little effects on the comparison between one-sided and two-sided primitives. Fig. 18 presents the results of micro-benchmarks using ConnectX-4 RNICs in an RoCE cluster. One-sided prim-

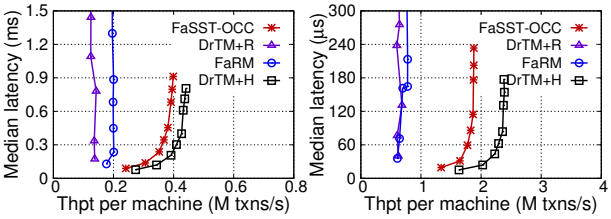


Fig. 17: *An end-to-end comparison of different designs for (a) TPC-C/no and (b) SmallBank using ConnectX-3 RNICs.*

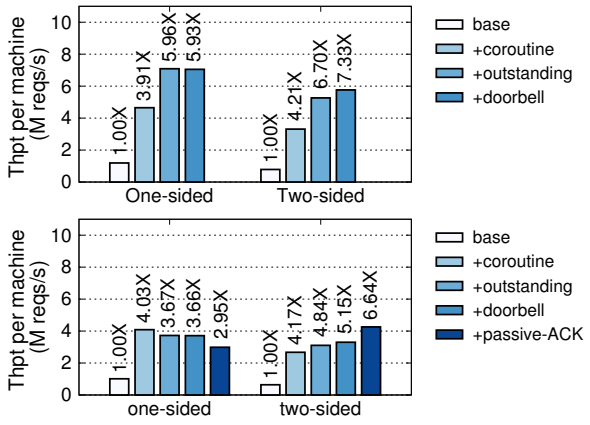


Fig. 18: *A comparison of one-sided and two-sided primitives using RoCEv2 NICs for multiple-object (a) reads and (b) writes with 256-byte payloads.*

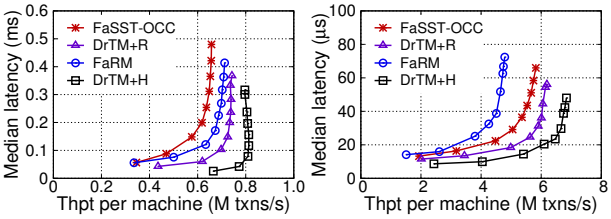


Fig. 19: *An end-to-end comparison of different designs for (a) TPC-C/no and (b) SmallBank using RoCEv2 NICs.*

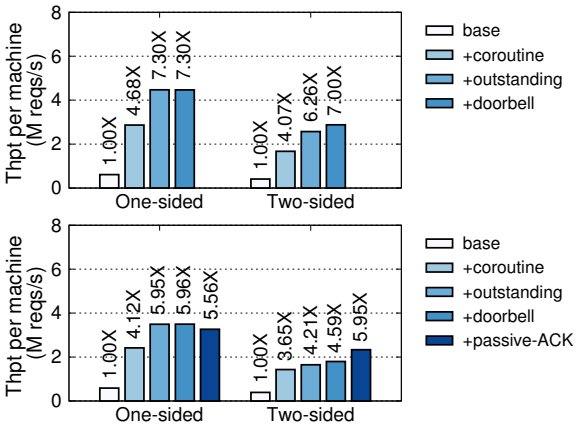


Fig. 20: *A comparison of one-sided and two-sided primitives using ConnectX-5 RNICs for multiple-object (a) reads and (b) writes with 256-byte payloads.*

itive still has better performance, except when two-sided one uses the passive-ACK optimization.

Fig. 19 further presents the end-to-end comparisons of prior designs using the RoCE cluster. We can achieve similar results as in §6. Note that both workloads achieve a better performance in this experiment. This is because both workloads use 2-way replication due to the restrictions of cluster size.

ConnectX-5 (CX5). Finally, we evaluate different systems using CX5, the later product of CX4. Due to the restrictions of RNIC (1 per machine), we utilize threads on the same socket for the experiments using CX5.

Fig. 20 presents the performance of primitive level analysis. One-sided primitives can achieve better performance, even when applying passive-ack optimization for two-sided primitive. This result is as we expected since Mellanox marks CX5 and CX4 as the same generation RNIC in its document (while CX3 is the previous generation RNIC). Fig. 21 further presents the end-to-end comparisons on `TPC-C/no` and `SmallBank` using CX5. The results are similar to results when using CX4.

8 Discussion

Trends, features, and extensions. Our studies focus on Mellanox ConnectX-4 RNIC. Previous generations of RNICs like ConnectX-3 yields slower performance of one-sided `READ`s. However, we have seen a trend that one-sided primitives become faster and more scalable in recent RNICs, from Connect-IB to ConnectX-4 to ConnectX-5. Further, new generation RNIC may introduce more features for one-sided primitives. For example, ConnectX-5 integrates one-sided `WRITE` with NVM [27]. This suggests an optimistic opinion about providing offloading features in modern data centers.

On the other hand, one-sided primitive still has many limitations due to the lack of expressiveness [51]. For example, it is not competent for complicated operations, like searching in a sorted store. Furthermore, one-sided primitive is unlikely to have orders of magnitude higher performance than messaging, because we have also seen a trend on providing fast messaging rate in later generation RNICs [28]. Hence, how to properly choose the right primitive is very important given a specific workload. This paper gives an example of how to optimize transactional processing with a combination of different primitives in a phase-by-phase way. The resulting system and insights may be reused for further studies.

Some proposed RDMA extensions, including the coherence of atomic operations, atomic object reads [9], and multi-address atomics [35], may provide further exploration spaces once being commercialized. We believe that there will be a continued line of research in this field with more new features, implementations and application domains.

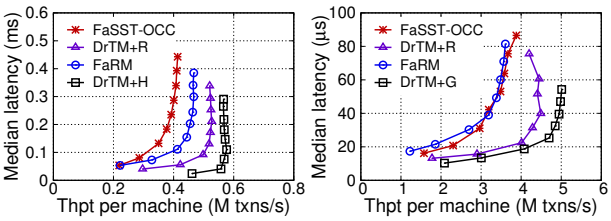


Fig. 21: *An end-to-end comparison of different designs for (a) TPC-C/no and (b) SmallBank using ConnectX-5 NICs.*

Emulating a large-scale RDMA cluster. Currently, we mainly focus on emulating massive RDMA connections in a rack-scale cluster, because QP cache misses will dominate the impact on the performance of various primitives. Consequently, we do not consider other scalability issues in a real large-scale RDMA cluster. For example, a large cluster has to use multiple layers of RDMA networking such as multiple switches or congestion control mechanism [60]. However, such aspects affect all network primitives instead of affecting only primitives. We plan to further validate our conclusion on a real, large RDMA-capable cluster in future.

9 Related Work

Existing RDMA optimizations. A set of optimizations on how to better leverage RDMA have been proposed. FaRM [10] proposes a set of techniques to mitigate cache pressure of RNIC, including using huge page to reduce page entries stored in RNIC and sharing QPs between threads to reduce the connections. HERD [16] first discovers the benefits of using UD QPs for messaging to improve performance and scalability. A recent guideline paper of RDMA [17] describes several optimizations on better leveraging RDMA features, including using doorbell mechanism to post a batch of requests. It also studies how low-level factors (e.g., payload inlining) impact the overall performance. FaSST [18] argues that UD, though unreliable as its name, has high reliability in modern datacenters because RDMA assumes a lossless link layer. Hence, UD QP is well suited for two-sided primitives. Finally, LITE [46] proposes a kernel indirection layer for RDMA which improves the scalability and programmability of RDMA. Many of such optimizations can be used cumulatively to improve performance. We apply most of them in our execution framework to make a fair comparison between one-sided and two-sided primitives, except for LITE. Because the optimization requires modifying the kernel and is not designed for our scenario.

Comparisons on RDMA primitives. Prior work has done valuable comparisons on different RDMA primitives [16, 10, 11, 18]. Our work continues such comparisons with a comprehensive study of both RDMA primitives and state-of-the-art optimizations. Further, we show that, even if one primitive performs better in micro-benchmarks, applications still need careful choices of RDMA primitives and optimizations to achieve the optimal performance.

Fast distributed transaction systems. We continue the line of research of providing fast distributed transactions [45, 7, 59, 31, 53, 51, 1, 24, 11, 57, 1, 54, 5, 18, 56]. Some systems leverage variants of OCC for consistency [24, 11, 5, 18], while others use algorithms such as 2PL [51] and SI [56] to handle workloads with more contentions. This paper uses OCC as an example to illustrate the effectiveness of a novel combination of RDMA primitives. We believe our insights on RDMA primitives may be applied to other concurrency control algorithms.

Other RDMA-enabled systems. A large number of systems have used RDMA features to improve performance. These include transaction processing systems [51, 11, 5, 18, 56, 50], key-value stores [30, 29, 16, 10, 40], distributed file systems [26, 38], consensus algorithms [34, 48] and graph processing systems [52, 36, 58]. Such systems also have different RDMA primitive choices according to their own demands. Our study may also inspire an optimal use of RDMA primitives on such systems.

10 Conclusion

We have presented a detailed analysis of how different choices of RDMA primitive affect the performance of transactional execution. Unlike previous studies, we compare different primitives and techniques using one well-optimized RDMA framework. This makes the comparison of techniques and primitives comparable and comprehensive. The main observations made by our study is that no single primitive is the winner all the time, even at different phases of transactional execution. We then propose a hybrid solution which uses the most appropriate primitive at each phase of transactions. This not only improves the throughput but also reduces the latency of transactions. Finally, our study gives hints about whether it is cost-effective to offload RDMA one-sided, or just use two-sided for easy porting. We hope this can stimulate and provide a guideline for future co-design with RDMA.

11 Acknowledgment

We sincerely thank our shepherd Marcos Aguilera and the anonymous reviewers in OSDI for their insightful suggestions, Hong Yang for sharing his experience on RDMA. This work is supported in part by the National Natural Science Foundation of China (No. 61772335, 61572314), the National Key Research & Development Program (No. 2016YFB1000500), the National Youth Top-notch Talent Support Program of China, and Singapore NRF (CREATE E2S2).

References

[1] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 245–262, New York, NY, USA, 2015. ACM.

[2] I. T. Association. Infiniband architecture specification. https://cw.infinibandta.org/document/dl/7859, 2015.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, July 2017.

[5] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154. ACM, 2010.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, 2012.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[9] A. Daglis, D. Ustiugov, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 6:1–6:13, Piscataway, NJ, USA, 2016. IEEE Press.

[10] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414. USENIX Association, 2014.

[11] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[12] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 202–215, New York, NY, USA, 2016. ACM.

[13] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, Jan. 2017.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, pages 289–300, New York, NY, USA, 1993. ACM.

[15] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM'14, pages 295–306. ACM, 2014.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, USENIX ATC '15, pages 437–450, Denver, CO, 2016. USENIX Association.

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[20] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic Storage-Performance Tradeoff in Data Stores. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Santa Clara, CA, Mar. 2016. USENIX Association.

[21] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[22] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[23] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM.

[24] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 71–86, New York, NY, USA, 2015. ACM.

[25] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, page 135, 2016.

[26] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[27] Mellanox Technologies. Nvme over fabrics standard is released, 2018. http://www.mellanox.com/blog/2016/06/nvme-over-fabrics-standard-is-released/.

[28] Mellanox Technologies. Products overview, 2018. http://www.mellanox.com/page/products_overview.

[29] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114. USENIX Association, 2013.

[30] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC'16, pages 451–464, Denver, CO, 2016. USENIX Association.

[31] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Berkeley, CA, USA, 2014. USENIX Association.

[32] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka. Telecom Application Transaction Processing (TATP) Benchmark. http://tatpbenchmark.sourceforge.net/, 2011.

[33] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 61–72, New York, NY, USA, 2012. ACM.

[34] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM.

[35] S. Raikin, L. Liss, A. Shachar, N. Bloch, and M. Kagan. Remote transactional memory, 2015. US Patent App.

[36] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 317–332, Berkeley, CA, USA, 2016. USENIX Association.

[37] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[38] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[39] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[40] T. Szepesi, B. Wong, B. Cassell, and T. Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads using rdma. In *First International Workshop on Rack-scale Computing*, 2014.

[41] M. Technologies. libmlx4 driver. http://www.mellanox.com/downloads/ofed/MLNX_OFED-4.0-1.0.1.0/MLNX_OFED_LINUX-4.0-1.0.1.0-ubuntu16.04-x86_64.tgz, 2017.

[42] The H-Store Team. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2018.

[43] The Transaction Processing Council. TPC-E Benchmark V1.14. http://www.tpc.org/tpce/.

[44] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/, 2018.

[45] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 1–12. ACM, 2012.

[46] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 306–324, New York, NY, USA, 2017. ACM.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[48] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 94–107, New York, NY, USA, 2017. ACM.

[49] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15. ACM, 2014.

[50] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 335–347, Santa Clara, CA, 2017. USENIX Association.

[51] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104. ACM, 2015.

[52] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[53] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 495–509. USENIX Association, 2014.

[54] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 279–294, New York, NY, USA, 2015. ACM.

[55] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[56] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[57] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. ACM.

[58] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 614–630, New York, NY, USA, 2017. ACM.

[59] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 276–291. ACM, 2013.

[60] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM'15, pages 523–536, New York, NY, USA, 2015. ACM.

Dynamic Query Re-Planning Using QOOP

Kshiteej Mahajan[1] Mosharaf Chowdhury[2] Aditya Akella[1] Shuchi Chawla[1]
[1]*University of Wisconsin - Madison* [2]*University of Michigan*

Abstract

Modern data processing clusters are highly dynamic – both in terms of the number of concurrently running jobs and their resource usage. To improve job performance, recent works have focused on optimizing the cluster scheduler and the jobs' query planner with a focus on picking the right query execution plan (QEP) – represented as a directed acyclic graph – for a job in a resource-aware manner, and scheduling jobs in a QEP-aware manner. However, because *existing solutions use a fixed QEP throughout the entire execution*, the inability to adapt a QEP in reaction to resource changes often leads to large performance inefficiencies.

This paper argues for *dynamic query re-planning*, wherein we re-evaluate and re-plan a job's QEP during its execution. We show that designing for re-planning requires fundamental changes to the interfaces between key layers of data analytics stacks today, i.e., the query planner, the execution engine, and the cluster scheduler. Instead of pushing more complexity into the scheduler or the query planner, we argue for a redistribution of responsibilities between the three components to simplify their designs. Under this redesign, we analytically show that a greedy algorithm for re-planning and execution alongside a simple max-min fair scheduler can offer provably competitive behavior even under adversarial resource changes. We prototype our algorithms atop Apache Hive and Tez. Via extensive experiments, we show that our design can offer a median performance improvement of 1.47× compared to state-of-the-art alternatives.

1 Introduction

Batch analytics is widely used today to drive business intelligence and operations at organizations of various sizes. Such analytics is driven by systems such as Hive [5] and SparkSQL [19] that offer SQL-like interfaces running atop cluster computing frameworks such as Hadoop [4] and Spark [59]. Figure 1 shows the key layers of data analytics stacks today. At the core of these systems are

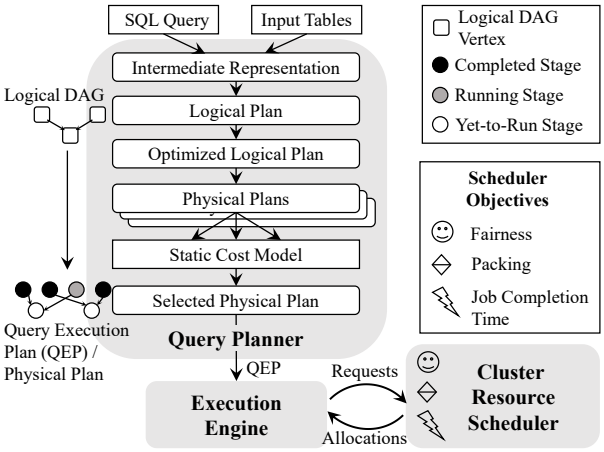


Figure 1: Traditional batch query execution pipeline.

query planners (QPs) , such as Calcite for Hive [3] and Catalyst for SparkSQL [19]. QPs leverage data statistics to evaluate several potential query execution plans (QEPs) for each query to determine an optimized QEP. The optimized QEP is a DAG of interconnected stages, where each stage has many tasks. An execution engine then handles the scheduling of these tasks on the underlying cluster by requesting resources from a scheduler. The scheduler allocates resources considering a variety of metrics such as packing, fairness, and job performance [35, 36, 61].

To improve query performance, existing works have primarily looked at optimizations limited to specific layers in the data analytics stack. Some of them [58, 61, 18, 34, 36, 35, 51] have focused on improved scheduling given the optimized QEP by incorporating rich information, such as task resource requirements, expected task run times, and dependencies. Others have considered improving the QP to take into account resource availability at query launch time (in addition to data statistics) to find good resource-aware QEPs [54, 55].

We argue that these state-of-the-art techniques fall short in *dynamic environments*, where resource availability can

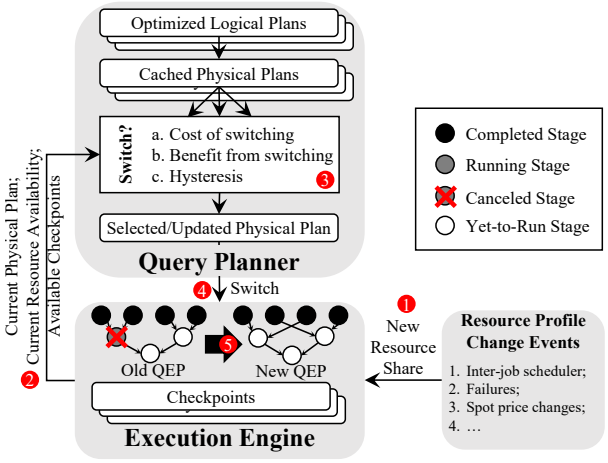


Figure 2: Dynamic replanning in action using QOOP. Omitted part of the query planner is similar to that of Figure 1.

vary significantly over the duration of a job's lifetime. This is because existing techniques are *early-binding* in nature – a QEP is pre-chosen at query launch time and the QEP's low-level details (e.g., the physical tasks, task resource needs, dependencies) are used to make scheduling decisions (which tasks to run and when). This fundamentally leaves limited options to adapt to resource dynamics. Our paper makes a case for *constant query replanning* in the face of dynamics. Here, a given job switches query plans during its execution to adapt to changing resource availability and ensure fast completion.

Dynamic resource variabilities can arise in at least two situations: (i) running multiple jobs on small private clusters, which is a very common use-case in practice [6]; and (ii) leveraging spot market instances for running analytics jobs, which is an attractive option due to the cost savings it can offer [45, 52, 62, 38]. We empirically study resource changes in these situations in Section 2.

To enable effective adaptation in these situations, we develop and analyze *strategies for query replanning*. We prove two basic results: (1) When dynamically switching QEPs, it is important for a query to potentially *backtrack* and forgo already completed work. Given imperfect knowledge of future resource availability, a query's performance can be arbitrarily bad without backtracking. (2) A *greedy algorithm* – which always picks a QEP offering the best completion time assuming current allocation persists into the future – performs well. We prove that the greedy algorithm has a competitive ratio of 4; the lower bound for any online algorithm is 2.

To realize the aforementioned replanning strategies in practice, we eschew the early binding in today's approaches. Instead, we propose a new system, QOOP, that has the following radically different division of labor and interfaces among the layers of analytics stacks (Figure 2):

- The cluster scheduler implements *simple* cluster-wide weighted resource shares and *explicitly* informs a job's execution engine of changes to its cluster share. The cluster share of a job is defined as the total amount of each resource divided by the number of active jobs. During a jobs's execution, our scheduler tracks a job's current resource usage – measured as the maximum of the fractions of any resource it is using – and allocates freed up resources to the job with the least current usage, emulating simple max-min fair sharing. Thus, the scheduler decouples the feedback about cluster contention – this helps queries replan and adapt – from task-level resource allocation, which is instantaneously max-min fair.

- When resource shares change *significantly*, the query planner compares a query's remaining time to completion based on its current progress against its expected completion time from replanning and switching to a different plan. It uses a model of task executions and available checkpoints in the execution engine to make this decision. It picks a better QEP to switch to (if one exists), and informs the execution engine of the new set of tasks to execute and existing ones to revoke.

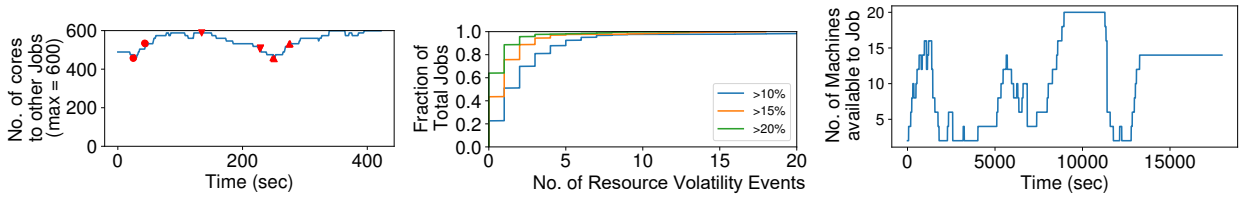
- The *execution engine* supports the query planner by informing it of the query's current progress and maintaining checkpoints of the query's execution from which alternate QEPs' computation can begin.

Overall, QOOP pushes complexity up the stack, out of cluster schedulers – where most of the scheduling complexity exists today – and into a tight replanning feedback loop between the query planner and the execution engine. We show that the resulting *late binding* enables better dynamic query adaptation.

We prototype QOOP by refactoring the interfaces between Hive, Tez, and YARN. Our evaluations on a 20-node cluster using TPC-DS queries show that QOOP's dynamic query replanning and simple scheduler outperform existing state-of-the-art static approaches. From a single job's perspective, QOOP *strictly* outperforms a resource-aware but static QP. For example, when resource profiles fluctuate rapidly, with high volatility, QOOP offers more than 50% of the jobs improvements of 1.47× or more; 10% of the jobs see more than 4× gains! We also use QOOP to manage the execution of multiple jobs on a small 20-node private cluster. We find that QOOP performs well on all three key metrics, i.e., job completion times, fairness, and efficiency, by approaching close to the individual best solutions for each metric.

2 Background and Motivation

In this section, we highlight multiple sources of resource dynamics in a cluster (§2.1), discuss the opportunities lost from not being able to switch a query's plan in response to resource dynamics (§2.2), and why the existing interfaces



(a) Jobs' view of cluster resources **(b)** Distribution of resource volatility **(c)** Number of machines reserved by a job in a spot market

Figure 3: Analysis of resource perturbations in a shared cluster and spot market. The gaps between each pair of the same symbols in (a) demarcate one resource volatility event.

between cluster schedulers, execution engines, and query planners make dynamic switching difficult (§2.3).

2.1 Resource Dynamics in Big Data Environments

Modern big data queries run in dynamic environments that range from dedicated resources in private clusters [25, 22] and public clouds [1] to best-effort resources put together from spot markets in the cloud [45, 52, 62, 38].

In case of the former, resources are arbitrated between queries by an inter-job scheduler [35, 30, 36, 61, 18]. As new jobs arrive and already-running jobs complete, resource shares of each job are dynamically adjusted by the scheduler based on criteria such as fairness, priority, and time-to-completion. Although in large clusters, such as those run by Google [25, 28] and Microsoft [22], individual job arrivals or departures have negligible impact on other jobs, most on-premise and cloud-hosted clusters comprise less than 100 machines [6, 9] and run only a handful of jobs concurrently. A 2016 Mesosphere survey [6] found that 96% of new users, and 75% of regular users use fewer than 100 nodes. A single job's arrival or completion in such scenarios can create large resource perturbations.

To better highlight resource perturbations in small clusters, we ran a representative workload on a 20-node cluster managed by Apache YARN. The cluster uses the Tetris [34] cluster scheduler, and it can concurrently run 600 containers at it's maximum capacity (1 core per container in a 600 core cluster). For our workload, we use the TPC-DS [12] workload, where jobs arrive following a Poisson process with an average inter-arrival time of 20 seconds. The average completion time per job is around 500s. We pick a job executed in the cluster and show it's view of cluster resources in Figure 3a. Specifically, we show the number of cores allocated (out of a maximum of 600) to all the *other* jobs running concurrently. During its lifetime, the job we picked experiences resource volatility – we call an x% increase or decrease in resource (number of cores in this case) over some period of time as an x% resource volatility. In Figure 3a, we identify 15% resource volatility within uniquely shaped red markers; e.g., the region between two solid red circles indicates one such 15% resource volatility. The job observes 3 such resource volatility events during its lifetime (identified within similarly shaped markers). To understand resource volatility as observed by different jobs for different resource volatility magnitudes (different values of x), in Figure 3b, we plot a CDF of the number of resource volatility events seen by each of the individual jobs in our workload for three values of x = 10%, 15% and 20%. We observe that almost 78% of the jobs experience at least one 10% resource volatility event during their lifetime, and 20% of the jobs see at least 4 resource volatility events of 10% or more.

At the other extreme, running jobs on spot instances – with their input on blob storage like Amazon S3 [2] – is becoming common because spot instances offer an attractive price point [45, 52, 62, 38]. However, cloud providers can arbitrarily revoke spot instances, which can cause perturbations in the number of machines available to a job. We now empirically examine the extent of such potential resource variations as experienced by a resource-intensive, batch job that runs for five hours. We use the spot-market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We also assume that the job has a budget of 5$/hour and that spot instances that were reserved at less than the current spot market price are taken away immediately. The spot instance prices typically update every minute. We use a simple cost-saving bidding strategy where the job progressively adds 2 spot instances every minute, provided the budget is not exceeded, by bidding at a price 5% over the current spot market price. Under such a bidding strategy and a budget of 5$, the maximum number of machines that the job gets is 20 and the minimum is 2. The number of spot instances available to the job over time is shown in Figure 3c. We make the following observations. First, the job experiences many perturbations in the number of machines, which is especially true with cost-saving bidding strategies. Second, the magnitude of perturbation is the largest around the 3

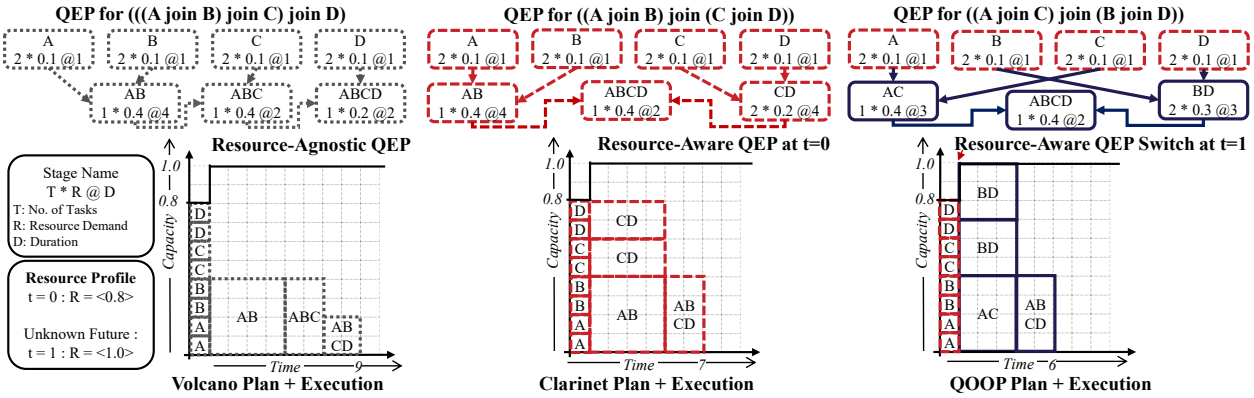


Figure 4: Comparison of query planners. The QEPs shown correspond to three different logical plans for the query - $A \bowtie B \bowtie C \bowtie D$. Volcano chooses the QEP on the left (as this plan has the least resource consumption) that completes at $t = 9$. Clarinet chooses an optimal resource-aware QEP (under static resources) at $t = 0$ that completes at $t = 7$; however, it ceases to be resource-aware when available resources change at $t = 1$. QOOP re-plans to switch to a new plan at $t = 1$ and completes the fastest at $t = 6$.

hour mark when the spot market instance price reaches a maxima of 0.5828$/hour and all but 2 machines are revoked. Finally, throughout the entire duration of the job, the job experiences 60, 53 and 40 resource volatility events of 10%, 15% and 20% respectively.

Other common sources of resource fluctuations include machine/rack failures, planned or unplanned upgrades, network partitions, etc. [21, 56].

2.2 Query Execution Today: Fixed Plans

Regardless of the extent of resource dynamics, existing approaches *keep the query plan fixed throughout the entire duration of a query's execution*. However, these approaches do vary in terms of what information they use during query planning and how they execute a query.

Resource-Agnostic Query Execution: A large number of today's data-analytics jobs are submitted as SQL queries via higher-level interfaces such as Hive [5] or Spark SQL [19] to cluster execution engines (Figure 1).

A cost-based optimizer (CBO) examines multiple equivalent logical plans for executing a query, and leverages heuristics to select a good plan, also called a query execution plan (QEP).[1] The QEP represents the selected logical plan and its relational operators as a *job* with a directed acyclic graph (DAG) of computation stages and corresponding tasks that will be executed by the underlying execution engine on a cluster of machines. Given the chosen QEP – also called the physical plan – the execution engine interacts with the cluster resource scheduler in a repeated sequence of resource requests and corresponding allocations until all the tasks in the physical plan of the job complete. *Crucially, the optimizer's heuristics are based on data statistics and not resource availability; thus, it is resource-agnostic.* An example is the Volcano query planner in Hive [5]. Figure 4 shows a Volcano-generated plan – a QEP corresponding to a "left deep" plan – that is preferred by the Volcano CBO based on data statistics.

Resource-Aware QEP Selection: Given the obvious inflexibility of resource-agnostic query optimization, some recent works [54, 55] have proposed resource-aware QEP selection. In this case, the CBO takes available resources into account before selecting a QEP and handing it over to the execution engine. While this is an improvement over the state-of-the-art, the execution engine still runs a fixed QEP even when resource availability changes over time. An example of a resource-aware planner is Clarinet [54]. As shown in Figure 4, the Clarinet plan is chosen based on the resources available at $t = 0$. When the resources change at $t = 1$, the static plan ceases to be the best.

Room for Improvement: Instead of sticking to the original resource-aware or -agnostic QEP throughout execution, one can find room for improvements by switching to a new QEP on the fly based on resource changes. For example, when the available resource increases at $t = 1$ in Figure 4, we can switch to a different join order – $(A \bowtie C) \bowtie (B \bowtie D)$ instead of $(A \bowtie B) \bowtie (C \bowtie D)$ – and further decrease query completion time.

Although this is a toy example, overall benefits of dynamic query re-planning improve with the complexity of query plans, magnitudes of resource volatility, and pathological fluctuations of resources due to unforeseen changes in the future (§6).

2.3 Scheduler Constraints on QEP Switching

Unfortunately, today's cluster schedulers and their interfaces with the execution engine and the query planner make resource-aware QEP switching challenging.

On the one hand, *existing schedulers provide little feedback to jobs about the level of resource contention in a cluster* – today, jobs simply ask the scheduler for re-

[1] Some optimizers consider a narrow set of resources, such as the buffer cache or memory, but ignore disk and network [5].

sources for runnable tasks and the scheduler grants a subset of those requests. Consequently, it is difficult for a job to know how to adapt in an informed manner to changing cluster contention or resource availability. One may think that jobs can infer contention by looking at the rate at which their resource requests are satisfied. However, such an inference mechanism can be biased by the resource requirements of the tasks in the currently chosen QEP instead of being correlated to the level of contention.

On the other hand, *scheduling decisions are tied to the intrinsic knowledge of job physical plan*. Schedulers are tasked with improving inter-job and cluster-wide metrics, such as fairness, makespan, and average completion time [30, 34, 27, 35, 36]. For example, DRF tracks dominant resources, which relies on the multi-dimensional resource requirements of physical tasks. Others [36, 34, 35] go further and combine resource requirements with the number of outstanding tasks and dependencies to estimate finish times using which scheduling decisions are made. The tight coupling of schedulers with pre-chosen QEPs constrains the scheduler to make decisions to match resources with the demands imposed by the pre-chosen QEP's tasks.

Overall, neither the job nor the scheduler has any way of knowing whether picking a different QEP with a very different structure and task-level resource requirements would have performed better – w.r.t. per-job or cluster-wide metrics – under resource dynamics.

3 QOOP Design

In this paper, we argue for breaking the constraints of fixed QEPs, and we make a case for continuous query re-planning by rethinking the division of labor between cluster schedulers, execution engines, and query planners. We first give an overview of our design (§3.1) and then present its three key components: a simple max-min fair scheduler (§3.2), an execution engine design to track additional states needed to speed up dynamic re-planning (§3.3), and a greedy QP that performs well with provable performance guarantees (§3.4).

3.1 Design Overview

The state-of-the-art approaches for improving query performance universally argue for pushing more complexity into the inter- and intra-job scheduling to achieve efficiency and improve job performance; by design, this prevents adaptation at the query level. Instead, to achieve replanning, we propose a significant refactoring. (1) We advocate having a simple max-min fair scheduler that effectively does "1-over-n" allocation of every resource across n jobs. (2) Jobs are informed as soon as their share changes due to changing n or machine/rack failures. (3) We push re-planning complexity *up the stack*, maintaining a *dynamic re-planning feedback loop* between the query planner and the execution engine: based on changes to the share, the planner – with help from the execution engine – determines if a better QEP exists and how to switch to it.

We choose this work division because each instance of an application framework today implements its own query planner and execution engine (e.g., both implemented in the Job Manager in case of frameworks using Apache YARN), whereas *all* jobs running in a cluster share the same centralized resource scheduler (i.e., the Resource Manager in Apache YARN). Our division of labor has the benefit of enabling many different applications with their intrinsic continuous re-planners to effectively run atop our simple cluster scheduler. For simplicity, our paper focuses just on re-planning batch SQL queries.

Figure 2 presents our architecture with the sequence of actions that take place on a resource change event: ❶ The cluster scheduler or the resource manager notifies the execution engine of its new resource share (§3.2). ❷ The execution engine, in turn, notifies the query planner of the current state, which includes the current QEP it is executing along with its progress, current resource availability it received from the scheduler, and the available set of checkpoints it is maintaining (§3.3). ❸ Given this information, the query planner must determine whether switching to a new plan is feasible (considering available checkpoints, cost of possible backtracking, and hysteresis) (§3.4). ❹ If the decision is yes, then it informs the execution engine of the new QEP. ❺ Finally, the execution engine will switch to the new QEP; if required, it will cancel some already-running stages and tasks.

Realizing dynamic re-planning raises a few key algorithmic questions. First, what is a good switching strategy when resources change? A simple and easy-to-implement choice is Greedy: i.e., always pick the QEP that offers the least estimated finish time assuming the new resource availability persists into the future. Does this offer good properties under arbitrary resource fluctuations? Second, switching from a QEP with partial progress to a new one that needs to be started from scratch necessarily wastes work. Is this "backtracking" necessary? In Section 4, we show that the simple Greedy approach performs well, and that backtracking is essential.

Before presenting the analysis, in Sections 3.3 and 3.4, we discuss key systems issues that arise in supporting greedy behavior with backtracking: How to estimate the relative runtimes of different QEPs? How to preserve work to support backtracking and leverage already computed work when switching to a new QEP? We start by outlining the functionality of our inter-job scheduler next.

3.2 Cluster Resource Scheduler

Our inter-job scheduler is simple (Pseudocode 1). For each job, our scheduler tracks the job's current (weighted) share in every resource dimension, i.e., the total fraction

of the resource that all currently running tasks of the job are using. The scheduler computes the *current share* of the job as the maximum of these fractions taken over all resources. When a resource is freed on a machine, our scheduler simply assigns it to the job with the lowest current share that can run on that machine, emulating simple instantaneous max-min fair allocation of resources across jobs, similar to [14, 30]. This is shown in lines 4–8. We are algorithmically similar to DRF, but differ in API. When resources become available DRF allocates to the job with least dominant-share; QOOP informs each job of its dominant share on resource-change events.

To enable re-planning, we introduce two changes to the interface between the scheduler and the execution engine. First, we do not require the execution engine to propagate the entire QEP to the cluster scheduler. Decoupling the QEP from the resources assigned to a job has the desirable property that the execution engine can change the QEP without affecting its fair share of resources, which is not the case for the state-of-the-art techniques [30, 36, 35].

Second, we introduce feedback from the cluster scheduler to the execution engine (line 9 in Pseudocode 1). Whenever the current cluster share of a job changes, the scheduler informs the job's execution engine. The cluster-wide fair-share informs each job of its minimum resource share given the current contention in the cluster. This acts as a *minimum resource guarantee* for the query planner when determining whether to re-plan in order to finish faster. In fact, any scheduler that can offer feedback in the form of an eventual minimum resource guarantee of resources to each job is compatible with QOOP.

3.3 Execution Engine

We discuss how job execution engine redesign can enable query re-planning, specifically backtracking.

Task Execution: Given a job QEP DAG (i.e., the output of a query planner), the execution engine executes tasks by interacting with the cluster scheduler while maintaining their dependencies. To determine the order of task execution, it can simply traverse the DAG in a breadth-first manner [59, 19] or use a multi-resource packing algorithm such as Tetris [34]. In QOOP, we use Tetris.

Checkpointing for Potential Switching Points: On any multi-resource update from the cluster scheduler, the execution engine relays the updated resource vector to the query planner to evaluate the possibility of switching to a different QEP. Determining whether to actually switch to a new QEP relies on multiple factors (§3.4). A major one is finding the suitable point(s) in the currently executing DAG to switch from. One may consider that switching from the currently executing stage or its immediate parent stage(s) would suffice. However, we prove in Section 4 that *backtracking* to ancestor stage(s) is essential for competitively coping with unknown future resource changes.

Pseudocode 1 Cluster Scheduler

```
1:  𝕁                          ▷ active jobs prioritized by lowest current share
2:  R⃗                          ▷ total cluster resource capacity
3:  U⃗                          ▷ consumed cluster resource portion

4:  procedure MaxMinFairScheduler
5:      pick first J ∈ 𝕁        ▷ triggered when R⃗ − U⃗ > 0⃗
6:      allocate demand D⃗_i ∈ J s.t. max_{i,m} D⃗_i · (R⃗_m − U⃗_m)
7:      update 𝕁
8:  end procedure

9:  procedure ResourceFeedback(Event 𝔼)
10:     𝕁 = 𝕁 ⊕ getJobChanges(𝔼)
11:     R⃗ = R⃗ ⊕ getResourceChanges(𝔼)
12:     fairShare = R⃗ / |𝕁|
13:     for all J_k ∈ 𝕁 do
14:         SendResourceFeedbackUpdate(J_k, fairShare)
15:     end for
16: end procedure
```

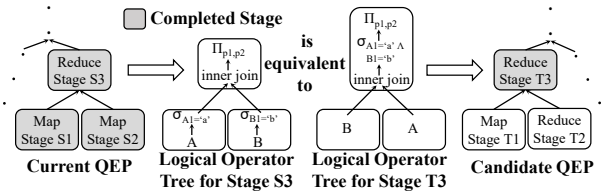


Figure 5: Progress propagation. First, we obtain logical operator trees for stages $S3$ and $T3$ from provenance. Stages $S3$, $T3$ are deemed equivalent as their logical operator trees are equivalent.

Consequently, QEP switching may not just re-plan the future stages of the query, but it requires the ability to checkpoint past progress and switch to a different QEP from an ancestor stage that was executed in the past. To enable this, the execution engine needs to checkpoint past progress for all the different QEPs it has executed thus far. Each checkpoint includes the intermediate outputs of completed tasks. Note that checkpointing of intermediate data is common in modern execution engines – disk-based frameworks write intermediate data to disks [25, 7], whereas in-memory frameworks periodically checkpoint to avoid long recomputation chains [59, 60]. QOOP can use this existing checkpointing.

Switching the QEP: The call back to the query planner (upon resource updates) is asynchronous. While the query planner is evaluating possible alternatives, the execution engine continues on with the current plan. When the query planner suggests a change, the execution engine revokes the resource requests for runnable tasks not belonging to the new QEP. Additionally, the execution engine may

abort running tasks not belonging to the new QEP. Thereafter, the execution engine resumes running tasks from the most-recent set of checkpoints for the new QEP.

3.4 Query Planner (QP)

In Section 4, we show that effective re-planning requires backtracking and that a greedy approach to re-planning results in a competitive online algorithm. Here, we present the details of how our query re-planner implements greedy re-planning by leveraging backtracking.

We introduce two key changes to the design of traditional QPs; neither requires extensive modifications. First, instead of discarding intermediate computations to explore and choose a particular QEP, we generate and cache several candidate QEPs. The cached QEPs later aid us in dynamic query re-planning. We also annotate each QEP with provenance, which consists of the original logical plan the QEP was derived from and the list of logical operators associated with each stage of the QEP. Figure 5 shows the provenance of each stage of a QEP.

Second, unlike traditional QPs [3, 54], our QP is made aware of the underlying resource contention to accurately predict runtimes for each QEP and greedily switch to the QEP with minimum completion time. To do so, we extend the interface between the QP and the execution engine so that the QP receives parameters to its dynamic cost model – the current resources available to the job (the share that the execution engine obtains from the cluster scheduler), the intra-job scheduling logic (packing), the progress of the current QEP and the available set of checkpoints.

Whenever the query planner receives a notification about resource changes from the execution engine, it triggers a cost-based optimization that involves predicting the completion times of all the QEPs and greedily switching to the QEP with earliest completion time. There are two steps to evaluate a particular QEP: progress propagation and completion time estimation.

Progress Propagation: To evaluate a candidate QEP, the QP first evaluates the work in the candidate QEP that is already done by the currently running QEP. It does so by identifying common work between the tasks of the candidate QEP, the running tasks of the current QEP, and the current set of checkpoints. We refer to this as *progress propagation*, and it is crucial in evaluating which candidate QEP to switch to and where to execute it from.

To identify common work as part of progress propagation, we identify equivalence between the stages of a candidate QEP and the set of checkpointed stages and the current running stages of the current QEP. To evaluate equivalence between two stages we generate the stages' logical operator trees using the provenance associated with each QEP. Two stages are deemed equivalent if their logical operator trees are equivalent. Equivalence of logical operator trees is evaluated using standard relational algebra equivalence rules. This is illustrated in Figure 5.

Completion Time Estimation: Next, we perform a simulated execution of the remaining tasks in the candidate QEP being evaluated (i.e., candidate QEP tasks whose work is not captured in the currently running QEP). Using the scheduling algorithm of the intra-job scheduler, i.e., Tetris, the remaining tasks are tightly packed in space and time given the current available resources. This yields an estimate for this QEPs completion time assuming that the current resource availability will persist in the future.

After evaluating the completion times of all candidate QEPs, query planner triggers a query plan switch if it finds a QEP that finishes faster than the currently running QEP. To avoid unnecessary query plan flapping, we add *hysteresis* by having a threshold on the percentage improvement of the query completion time – a query plan switch is triggered only if improvements exceed this threshold.

In case of a switch, the query planner sends the new QEP to the execution engine. This QEP is modified from its original form so that the DAG now contains the checkpoints as input stages, marks the running stages it shares with the running stages of the current QEP, and identifies the dataflow from these to the remaining stages.

4 Analysis

We now present analysis of the query planner (QP; Section 3.4). Each query has several alternative query execution plans (QEPs). We motivate the choices made in the query replanning algorithm regarding *why, when and which* QEP to switch to during the execution of a query in response to the resource allocations made by the scheduler to the query. This is an online algorithm since it operates without the knowledge of future resource allocations. We analyze the performance of our online algorithm in the form of its competitive ratio. Our goal is to argue that our online algorithm performs well no matter the sequence of resource allocations made to the query. We will compare our online algorithm's performance against an hindsight optimal (a.k.a. offline) algorithm which chooses the single best QEP knowing the entire sequence of resource allocations made to the query. The competitive ratio is the ratio of the performance of the online algorithm to that of the hindsight optimal (a.k.a. offline) algorithm. We provide a precise measure of comparison shortly (Section 4.3).

4.1 Notation and Assumptions

Notation: We represent each QEP as $a \times b$. This denotes a QEP with a bag of b tasks, each task needing a resource-units (e.g., number of cores) and each task completing in 1 step. The total work for this QEP is denoted by w and is equal to ab.

Assumptions: For the upper and lower bounds on performance, we assume an adversarial scheduler that can look at the algorithm's choices in the previous steps and

change future resource allocation in a worst-case manner. We require that the QP has the ability to *backtrack* a QEPs execution i.e., the QP can checkpoint each completed task in a QEP and any completed task need not be re-executed when the QP decides to switch back to and resume the execution of that QEP. We also assume that backtracking does not incur any overheads; in other words that our analysis ignores system-level costs (time spent and compute/memory used) in writing checkpoints and reading from checkpoints during a QEP switch.

4.2 Motivating Example

We motivate *why* QEP switching and specifically backtracking is necessary to obtain a bound on the performance of our online algorithm.

Our toy examples, with large work-differences in QEPs, serve to show that if the online algorithm does not make good decisions then its performance can become unboundedly worse.

Example 4.1.
QEP switching is necessary. *Consider a query with two QEP choices: the first one being* 2×2 *and the second one being* 1×100*. Suppose that the scheduler starts by giving the query* 2 *resource-units in the first step. We also suppose that the query cannot switch QEPs.*
CASE-1: *If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step. With this allocation, the optimal choice would be to run the* 2×2 *QEP, finishing in two steps and performing only* 4 *units of work. The online algorithm instead performs* 100 *units of work if it continues to use the* 1×100 *QEP.*
CASE-2*: On the other hand, if the query starts running the* 2×2 *QEP, the scheduler switches to a resource allocation of* 1 *resource-unit second step and onwards. Now the* 2×2 *QEP is stalled. Unless the algorithm switches to the* 1×100 *QEP, it is unable to finish.*
QEP backtracking is necessary. *Backtracking helps* avoid stalling, ensures fast completion, and bounds wasted work. *We continue the previous example. As before, the scheduler continues to be adversarial. It allocates* 1 *and* 2 *resource-units in the next step whenever the query is executing* 2×2 *and* 1×100 *QEP in the current step, respectively. Also, we now suppose that the query has the ability to switch QEPs but not backtrack i.e., no ability to checkpoint and resume QEPs from checkpoint.*
We continue from where we left-off in the previous example i.e., CASE-2 *where the query is executing the* 2×2 *QEP and the scheduler allocates* 1 *resource-unit in the second step. With the ability to switch, to avoid stalling, the query switches to the* 1×100 *QEP in the second step. Without backtracking, the query has to restart execution of* 1×100 *QEP from the beginning. Now on switching to the* 1×100 *QEP, the adversarial scheduler gives the query* 2 *resource-units third step onwards. This leads us back to* CASE-1*. If the QEP continues with the* 1×100 *QEP it leads to* slower completion.
If instead the query switches back to 2×2 *QEP in the third step, without backtracking the QEP restarts execution from the beginning and the adversarial scheduler gives the query* 1 *resource-unit fourth step and onwards. This is* CASE-2 *all over again. We can now see that, without backtracking, the query flips between* CASE-1 *and* CASE-2 *and* stalls infinitely with unbounded wasted work.
Even if the query decides to limit wasted work by stopping the switch to 2×2 *QEP, complete execution of* 1×100 *QEP to completion leads to* 100 *units of additional work and* 100 *additional steps. This leads to* slower completion *as in* CASE-1.
If the query could backtrack – we would have only one additional task to run from the 2×2 *QEP in the third step and the query would complete execution in the third step with just* 1 *units of wasted work.*

4.3 Competitive Ratio

A natural way to compare the performance of our algorithm against the hindsight optimal algorithm is to compare the time each algorithm takes to complete the query. As the next example shows, this is not a meaningful comparison, because the scheduler has the power to starve the online algorithm after a single bad choice.

Example 4.2. Starvation. *Consider the above example again. As before, the scheduler starts by giving the query* 2 *resource-units in the first step. If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step, and then gives no more resources to this query in subsequent steps. Regardless of whether the query continues running the* 1×100 *QEP or switches to the* 2×2 *QEP in the second step, the query is unable to finish the work and stalls. Its completion time is unbounded. With the same allocation of resources, the hindsight optimal algorithm could have finished the query by just running the* 2×2 *QEP.*

On the other hand, say the query starts running the 2×2 *QEP and the scheduler gives* 1 *unit resource for the next* 99 *steps and then gives no more resources. Once again no matter what the online algorithm does, it cannot complete the query. However, the hindsight optimal algorithm would have been able to complete the query given these resources.*

In each of the cases in the above example, the scheduler could stall the query for an unlimited time, whereas the hindsight optimal algorithm terminates in bounded time. In order to allow for some wasted work due to the online nature of the algorithm, the scheduler must provide more resources to the online algorithm than just the minimum necessary for the hindsight optimal algorithm.

Pseudocode 2 Online Query Planning Algorithm

Input n QEPs, $a_i \times b_i$, with $a_1 < a_2 < a_3 < \cdots < a_n$

1: Let $w_i = a_i b_i$ denote the total work of QEP i.
2: **for all** $i \in [n]$ **do**
3: **if** $w_i > \frac{1}{2} w_{i-1}$ **then** remove QEP i from the list.
4: **end if**
5: **end for**
6: At every step, given the current resource allocation a, consider all QEPs with $a_i \leq a$. Of these, run the QEP with the least remaining processing time, breaking ties in favor of the QEP with the smallest a_i.

To formalize this, we will compare the completion time of the online algorithm to that of an hindsight optimal algorithm that is required to perform extra work.

Definition 4.1. Competitive Ratio. *We say that an online QEP selection algorithm achieves a competitive ratio of α if for any query and any sequence of resource allocations, the completion time achieved by the online algorithm is at most equal to the completion time of an offline optimum that runs α back-to-back copies of the query.*

We note that α above does not have to be an integer.

4.4 Bounds for the Competitive Ratio

We show that no online algorithm can achieve a competitive ratio < 2. Proofs for the theorems below can be found in extended version of QOOP [47].

Theorem 4.1. *No online query planning algorithm can achieve a competitive ratio of $2 - \varepsilon$ for any constant $\varepsilon > 0$ when the resource allocation is adversarial.*

Our query planning algorithm corresponding to the simplifying assumptions in Section 4.1 is formally described above. It is greedy and at every step runs the QEP with the least remaining completion time with the assumption that the resource allocation persists forever. Also, it is "lazy" as it switches QEPs only when the resource allocation changes. Our overall approach in Sections 3.4 and 3.3 is a generalization of this algorithm for complex queries.

We prove that this algorithm is competitive:

Theorem 4.2. *The online greedy query planning algorithm described above achieves a competitive ratio of 4. Further, if the QEPs satisfy the property that every pair of QEPs is sufficiently different in terms of total work, in particular, $w_i \leq \frac{1}{2} w_{i-1}$ for all $i > 1$, then the competitive ratio is ≤ 2, matching the lower bound.*

We note that constant competitive ratio implies that the performance of our online query planning algorithm is independent of the nature of workloads or the environment.

5 Implementation

Implementation of QOOP involved changes to Calcite [3], Hive [5], Tez [7], and YARN [53]. QOOP's implementation took ~13k SLOC. The majority of our changes were in Tez mostly devoted to dynamic CBO module we elaborate upon shortly.

Hive and Calcite: Hive uses the Volcano query planner implemented in Calcite to get a cost-based optimized (CBO) plan. We add the ability to cache several logical plans in Calcite during its plan evaluation process and make changes to Hive to fetch multiple physical plans (i.e., Tez QEPs). Also, we make changes to annotate each QEP with provenance—the set of logical relational operators associated with each stage of the QEP. We widened the RPC interface from Hive to Tez, to push multiple QEPs to Tez as part of a single job.

Tez and Yarn: To enable dynamic query plan switching we added modules to Tez that are responsible for (i) accounting checkpoints to enable backtracking; (ii) dynamic cost-based optimization to make Tez QEP switching decisions; (iii) runtime QEP changes to realize QEP switching; and (iv) the RPC mechanism from YARN to Tez to give resource feedback (i.e., resource updates about the dynamic "1-over-n" share of resources).

Any resource change event from YARN triggers our dynamic CBO module that evaluates all QEPs. This module first propagates progress using provenance and estimates completion time of each QEP via simulated packing in the available resource share (§3.4). Our CBO relies on estimates of tasks' resource demands—CPU, memory, disk, and the network—and their durations. Peak resource estimates are based on prior runs for each QEP. We use these peak resource estimates to decide the container request sizes for tasks in the currently executing QEP.)

Checkpointing for backtracking and runtime changes to the QEP involve changes to the QEP, Vertex, and Task state machines in Tez. All checkpointing state is maintained at the Tez QEPAppMaster—which keeps the file handle of task output after every task completion event. For QEP switching, we added the `SWITCHING` state to the QEP state machine. On a resource change event a QEP is forced from `RUNNING` to `SWITCHING`. Any running tasks of the QEP continue running in this state but the launch of any new vertex (and hence its tasks) is prevented in this state. The QEP switches to `RUNNING` state after, if at all, QEP switching happens. During a QEP switch, the set of runnable Vertices is re-initialized to those from the new QEP. The Vertex definition is changed so that the inputs for the tasks spawned by any runnable Vertex points to the appropriate checkpoint.

6 Evaluation

In this section, we evaluate QOOP in situations with varying degrees of resource variabilities. We examine both

the performance of an individual query using QOOP's replanning as well as overall performance when multiple queries run atop QOOP.

We start by studying the execution of a single job, subjecting it to real resource change events or *resource profiles*. Specifically, in these *micro-benchmarks*, our focus is on answering the following key question: *does QOOP's dynamic query re-planning improve a job's completion time when compared to static, early-binding approaches?*

Next, we evaluate the key system components of QOOP – backtracking, overheads of QEP switching, robustness to errors in the task estimates, and hysteresis.

Finally, we consider a small private cluster, where QOOP is used to manage the execution of multiple jobs. We evaluate QOOP by running multiple jobs on the testbed, wherein job arrivals and completions can lead to large resource perturbations. This *macro-benchmark* addresses the question: *does QOOP's simple cluster scheduler and dynamic query re-planner approach improve system-wide objectives when compared against systems with complex schedulers and static query planners?*

6.1 Experimental Setup

Workloads: Our workloads consist of queries from the publicly available TPC-DS [12] benchmark. We experiment with a total of 50 queries running at a scale of 500, i.e., running on a 500GB dataset. [2] For micro-benchmarks, we focus on the perspectives of individual queries. For macro-benchmarks, each workload consists of jobs drawn at random from our 50 queries and arriving in a Poisson process with an average inter-arrival time of 80s. [3]

Cluster: Our testbed has 20 bare-metal servers – each machine has 32 cores, 128 GB of memory, 480 GB SSD, 1 Gbps NIC and runs Ubuntu 14.04. For micro-benchmarks, we evaluate QOOP under different realistic resource profiles, as elaborated later in this section. In such experiments, we provide as much resources from the cluster to each job over time as dictated by the resource profile. Specifically, whenever there is an increase in the amount of resources in the resource profile we make available to the job corresponding number of containers, whereas whenever there is a decrease in the amount of resources in the resource profile we immediately revoke equivalent number of containers and fail any tasks running on them.

For macro-benchmarks, we run our entire collection of jobs across the entire cluster. At its maximum capacity, the cluster can run 600 tasks (containers) in parallel.

Baselines: In micro-benchmarks, we compare QOOP's query planner against static query plans obtained from the *Clarinet QP*, which is a resource-aware QP implemented in Hive [54] that improves upon Volcano. We only compare against *Clarinet QP* as it outperforms Volcano. We adapted Clarinet to our setting to choose a QEP that minimizes completion time using resource estimates just before query execution begins. It represents the performance upper-bound of fixed-QEP approaches.

In macro-benchmarks, we compare QOOP – dynamic query planner on top of our simple max-min fair scheduler – against the following approaches on the three system-wide objectives of fairness, job completion time, and efficiency: (1) *DRF*: The default DRF multi-resource fair scheduler [30] in conjunction with Hive's default Volcano QP; (2) *Tetris*: A multi-resource packing scheduler [34] with Volcano; (3) *SJF*: Shortest-Job-First scheduler [27] with Volcano; (4) *Carbyne*: A meta-scheduler that leverages DRF, Tetris, and SJF [35] with Volcano; (5) *DRF+Clarinet*: DRF with the Clarinet QP [54]; (6) *Carbyne+Clarinet*: Carbyne scheduler with Clarinet QP.

These reflect combinations of query planners that differ in whether they are resource-aware with schedulers that differ in the complexity of information they leverage in making scheduling decisions.

Metrics: Our primary metric to quantify performance improvement using QOOP is improvement in the average *job completion time (JCT)*: $\frac{\text{(Average) JCT of an Approach}}{\text{(Average) JCT of QOOP}}$

Additionally, in multi-job scenarios, we consider *Jain's fairness index* [42] to measure fairness between jobs, and *makespan* (i.e., when the last job completes in a workload) to measure overall resource efficiency of the cluster.

6.2 QOOP in Micro-Benchmarks

QOOP has two core components: the dynamic query replanning logic for a single query (Sections 3.3 and 3.4), and the simple cross-job cluster wide scheduler (Section 3.2). We study the two separately, with this section focusing on the former using micro-benchmarks.

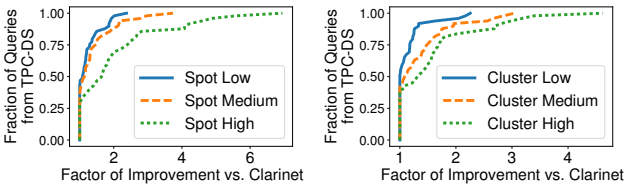
Specifically, in these micro-benchmarks, we ask: given a certain resource change profile, how well does a single query perform from using QOOP's query replanning algorithms? We study QOOP under two classes of resource change profiles, *spot instances* and *cluster resources* .

6.2.1 Spot Markets Resource Profiles

We obtained a 5-hour spot market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We infer the resource profile for the spot market price trend by applying the bidding strategy described in Section 2. We then divide the entire resource profile into "low", "medium", and "high" regions by time. To do so, we divide the entire resource profile into 10 minute regions and calculate the maximum increase or decrease in the resources in this 10

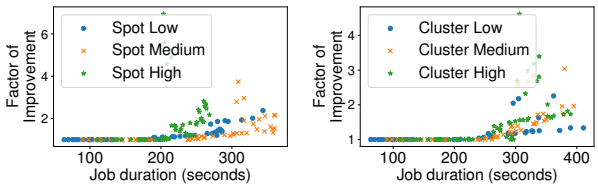
[2] We cached plans obtained while exploring QEPs in the Volcano planner, and retained plans with significant differences in cost according to Volcano's cost model. We used the first 50 TPC-DS queries that gave the most number of QEP alternatives.

[3] Google cluster trace [8] analysis on 20-machine sets yielded an average job inter-arrival time of 80s.



(a) Spot market **(b)** Shared cluster

Figure 6: Improvements using QOOP w.r.t. Clarinet under resource variations observed in different resource profiles.



(a) Spot market **(b)** Shared cluster

Figure 7: Improvements vs. job durations using QOOP w.r.t. Clarinet under different resource profiles.

minute region. We call an x% increase or decrease in resource in atleast one of the resource dimensions (compute or memory) over some period of time as an x% resource volatility. If the maximum resource volatility in resources is less than 10% then we classify this region as "low", if it is between 10% and 20% then we classify this region as "medium", if it is greater than 20% then it is classified as "high". We also refer to these as having "low", "medium" and "high" resource volatility . We then run each of our 50 TPC-DS queries individually against each of these three resource profiles – "low", "medium", and "high" – using both QOOP and Clarinet. For each query run with a particular resource profile type, we pick 10 different randomly selected regions of that particular profile type and report the mean from these 10 runs.

We plot the CDF of QOOP's improvements over Clarinet for the three resource profiles in Figure 6a. We see that QOOP *strictly* outperforms Clarinet, with its gains improving with increasing resource volatility – overall, 58%, 62% and 66% of the jobs experience faster completion times in each of "low", "medium", and "high" profiles, respectively. Median improvements for the "low", "medium" and "high" profiles are respectively 1.08×, 1.11× and 1.47×. For "high" profiles, 10% of jobs see gains $> 4\times$! We also note that 34% of the jobs show no improvements over Clarinet even with "high". On further analysis, we found that these jobs are queries in the TPC-DS workload that are either (i) less complex queries with lesser number of joins, or (ii) queries with short durations. Less complex queries may lack attractive alternative QEPs, whereas short queries may miss out on resource perturbations. This limits opportunities for re-planning and improvement. We dig deeper into these issues later in this section.

6.2.2 Shared Cluster Resource Profile

Similar to the spot market scenario, we generate three different resource profiles for the shared cluster scenario described in Section 2. Following a similar methodology, we identify "low", "medium" and "high" resource volatility periods, and we run each of the 50 queries.

As before, we plot the CDF of QOOP's improvements in Figure 6b. We see the trends similar to that of the spot market trace – overall, 56%, 58% and 60% of the jobs complete faster in "low", "medium", and "high" profiles, respectively. The median improvements in the three profiles are 1.08×, 1.11× and 1.20×, with higher performance improvements in greater resource volatility scenarios; for the "high" profile, 10% of jobs see gains $> 3.3\times$.

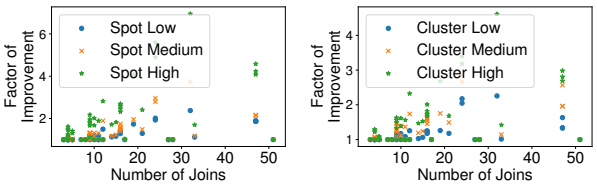
In both the spot instance and cluster profiles, gains are higher for profiles with higher volatility. In other words, QOOP's dynamic replanning is most effective relative to static query plans when resource volatility is at its highest. Also, the improvements for spot market and shared cluster, while similar for "low" and "medium", differ on the "high" resource profiles. We attribute this to spot market "high" resource profiles experiencing 7% larger magnitudes of resource changes at median than that of the shared cluster.

6.2.3 Delving into Improvements

Next, we take a deep dive into the aforementioned scenarios to understand when QOOP offers the greatest/least improvements. We study the impact of job duration, complexity, and the number of QEP switches that occur.

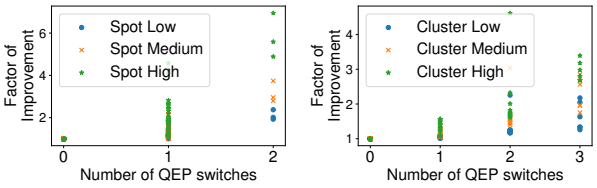
Job Durations vs. Observed Gains: The improvements in per-job performance due to QOOP as a function of job duration is shown in Figures 7a and 7b for the spot market and cluster resource profiles, respectively. Both figures also show results for the "low", "medium" and "high" volatility profiles using different-colored dots. In both cases, QOOP's benefits increase with increasing job durations. This is because longer jobs receive more opportunities for switching query plans and the comparative overhead of a switch of a longer job is smaller w.r.t. its completion time. Nevertheless, some shorter jobs benefit from QOOP in case of higher resource volatility.

Job Complexity vs. Improvement: Figures 8a and 8b show improvements obtained with QOOP as we increase query complexity for the spot market and cluster profiles, respectively. We measure query complexity in terms of the number of join operations in the query. We make two observations. First, increased query complexity generally correlates with increased gains. This is because the number of alternate query execution plans is higher with a greater number of joins. Second, keeping complexity



(a) Spot market **(b)** Shared cluster

Figure 8: Improvements vs. query complexity (number of joins) using QOOP w.r.t. Clarinet under various resource profiles.



(a) Spot market **(b)** Shared cluster

Figure 9: Improvements vs. number of QEP switches using QOOP w.r.t. Clarinet under various resource profiles.

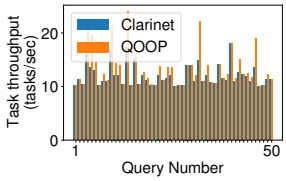


Figure 10: Improvements w.r.t. Clarinet vs. number of QEP switches in spot market.

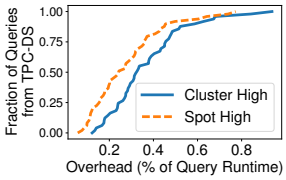


Figure 11: Overheads due to QEP switches measured as % of a job's completion time.

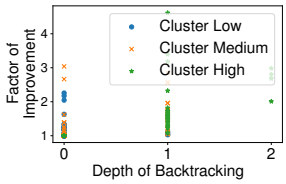


Figure 12: Improvements vs. depth of backtracking.

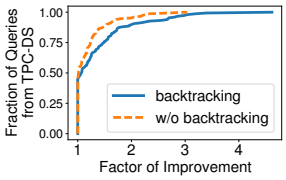


Figure 13: Improvements with and without backtracking.

constant, higher volatility results in the highest factor of improvement (as indicated above).

QEP switches vs. Improvement: Figures 9a and 9b shows the trend between improvements and number of runtime QEP switches. First, we see that an increase in the number of query execution plan switches correlates with increased gains. Second, keeping the number of switches constant, higher volatility results in the highest factor of improvement. In general, the greater flexibility a query intrinsically has in terms of multiple alternate plans together with the flexibility QOOP offers in switching to these plans results in a higher degree of improvement.

Task Throughput: Finally, we consider how fast QOOP helps the query complete tasks over time. We measure task throughput as the average number of tasks of the job executed per second; higher implies better utilization. In Figure 10 we show the task throughput of QOOP and Clarinet across queries. The number of tasks per second in the case of QOOP exceeds Clarinet by $\sim 24\%$ in the average case. Further analysis showed that an increase in the number of resources available leads QOOP to switch to query execution plans that favor more parallelism (i.e., "bushy" joins) and contributes to increased utilization.

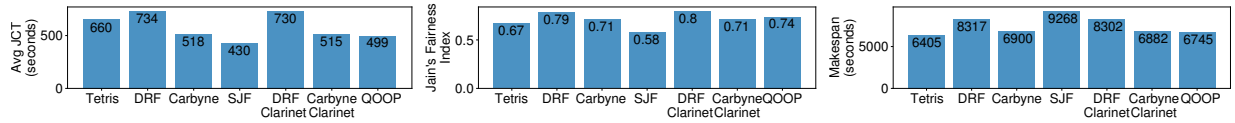
6.3 Impact of Various QOOP Features

In this section, we study the effect of different aspects of QOOP on the performance observed by a single query.

Backtracking: Figure 12 shows the relationship between improvement factor and the depth of backtracking in a shared cluster setting with different resource profiles. We observe that the depth of backtracking (i.e. the maximum distance of the vertex in the switched-to QEP from any running/completed vertex in the current QEP) increases with the magnitude of resource change events. 5.7% of all the runs experience a backtracking to two stages deep in the past and is triggered only by "high" volatile resource profile. 85.3% of the experimental runs with "low" volatile resource profile experienced no backtracking. We observe similar results for spot market setting. Figure 13 shows the CDF of factor of improvement w.r.t. Clarinet with and without backtracking turned on for the runs of all our TPC-DS queries when run under shared cluster resource profiles. We observe that when backtracking is turned on QOOP yields higher factor of improvement as backtracking finds better QEP switches.

Overhead of QEP switch: Figure 11 shows the overheads of QOOP in the shared cluster and spot market settings. We measure overhead as the time a job spends in switching to alternate QEPs as a percentage of total job time. The overheads in the shared cluster are 0.15% higher than in the spot market setting. This is because of the higher number of overall QEP switches when a job runs in a shared cluster – also shown in Figure 9b. On the whole, however, the overhead due to QEP switching has negligible impact ($< 1\%$) on overall job performance. The overall QEP switching overhead is low as hysteresis prevents unnecessary QEP switching and the absolute number of QEP switches in a job is low – at most 3 as shown in Figures 9a and 9b.

Robustness to Error: Figure 15 shows QOOP's robustness to error in the estimates of task resource demands and durations. We introduce X% errors in our estimated task demands and durations. Specifically, we select X in [-25, 25] as suggested by prior work [35], and increase/decrease resource demands by $task_{newReq} =$



(a) Job Performance **(b)** Inter-Job Fairness **(c)** Cluster Efficiency

Figure 14: Comparison of performance, fairness, and cluster efficiency of QOOP w.r.t. existing solutions. Higher values are better for fairness, whereas the opposite is true for the rest.

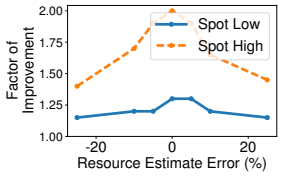


Figure 15: Error robustness in QOOP.

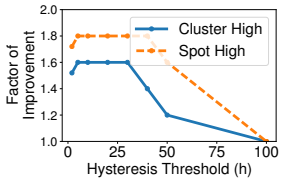


Figure 16: Effect of hysteresis on improvements.

$(1+X/100)*task_{origReq}$, and task durations change similarly. We study these errors in simulation against low and high volatile spot market resource profiles. We observe even at the highest error rates of $\pm 25\%$, QOOP offers substantial performance improvements (e.g., $1.4\times$ for the high volatile profile). For low volatile resource profile, QOOP is more robust to estimation errors: at 25% error rate, the performance improvement is $1.18\times$ compared to $1.25\times$ at no error. However, mis-estimations are costly at high volatility: errors $\geq 10\%$ cause performance improvement to drop 33% or more; nevertheless, QOOP's performance is always better than Clarinet.

Hysteresis: Figure 16 shows the effect of our hysteresis threshold (h) on the improvements. In QOOP, hysteresis prevents QEP switch unless there is an $h\%$ improvement in the estimated job completion time. We experiment with different values of h for "high" resource profiles for both spot market and shared cluster. A very high hysteresis threshold prevents switching, hurting performance. By definition, setting hysteresis parameter (h) to 0 causes more QEP switching (because of lower thresholds for QEP switching) and hence slightly higher overhead; we still see positive gains. However, for both traces, we observe that there is a wide range of h values where the factor of improvement sustains it's peak. This means that QOOP has flexibility to choose h; any value in the 10% - 25% range offers good performance at low switching overhead.

6.4 QOOP in Macro-Benchmarks

So far we have evaluated QOOP in offline, micro-benchmarks against the Clarinet QP with an aim to understand its query re-planning capabilities. In a real cluster, however, jobs arrive in an online fashion. Consequently, the impact of scheduling on job performance and its interplay with the QP become important.

In this section, we evaluate QOOP in an online setting in our shared cluster, where 200 TPC-DS jobs – randomly drawn from the 50 TPC-DS queries – arrive following a Poisson process with an average inter-arrival time of 80 seconds (Figure 14). As mentioned earlier, we compare QOOP against a wide range of solutions in both categories: scheduling and query planning. On the one hand, we consider a variety of scheduling solutions such as DRF, Tetris, SJF, and Carbyne that focus on objectives ranging from simple fairness (QOOP) to improving multiple goals. On the other hand, we consider QPs that range from static resource-agnostic planning (Volcano in Hive) to resource-aware early-binding (Clarinet) to QOOP's late-binding re-planner. Finally, in addition to focusing only on job completion time, which is useful only to individual jobs, we consider cluster-level metrics such as fairness (measured in terms of Jain's fairness index [42]) and efficiency (measured in terms of makespan).

Job Performance: First, we observe that QOOP significantly improves the average JCT w.r.t. simple state-of-the-art solutions (Tetris, DRF) and comes closest to the average JCT of SJF (Figure 14a). Furthermore, it outperforms the state-of-the-art in complex scheduling and QP alternatives: Carbyne and DRF+Clarinet, respectively. Only by combining two complicated solutions (Carbyne+Clarinet), the state-of-the-art can come close to QOOP. This suggests that the inflexibility of the current interfaces have tangible costs and overcoming them requires introducing complexities at every layer of the analytics stack.

Fairness Between Jobs: If performance were the only concern, one could get away with simply using SJF instead of using the complex alternatives or QOOP. However, performance and fairness have a strong tradeoff [35] as shown in Figure 14b – SJF has the worst fairness characteristics! We observe that while DRF and DRF+Clarinet are the most fair solutions, QOOP comes the closest to them while ensuring almost $1.5\times$ smaller average JCT.

Cluster Efficiency: Finally, Tetris performs well in its goal of packing tasks better and achieving high efficiency (Figure 14c), but QOOP again comes the closest to Tetris.

Overall, QOOP improves all three metrics – performance, fairness, and efficiency – over complex state-of-the-art solutions or combinations thereof, and achieves

these benefits using a simple scheduler with a dynamic, resource-aware QP that can re-plan queries at runtime.

7 Related Work

Other Applications: Although we focus SQL queries, the high-level principle of designing dynamic resource-aware plan switching can be applied to many other applications. This is because many frameworks use query planners to create execution plans for workloads, e.g., in machine learning [32, 44, 49], graph processing [33, 46, 48], approximation [15, 10] and streaming [60, 11, 13, 16, 50].
Query Planners in Big Data Clusters: Query planning is a well-trodden research area with numerous prior work [37]. We restrict our focus on query planners designed for distributed big data clusters that fall into two broad categories: those who plan a query in a resource-agnostic manner [3, 19] and those who are resource-aware [54]. Both, however, result in static query plans throughout the execution of a job. There is a massive body of work on adaptive query processing [26] in the context of traditional (single-machine) database systems. We focus on big data analytics in multi-node clusters.
Execution Engines: Execution engines take job DAGs and interact with the cluster scheduler to run all the tasks of each job until its completion. Examples of popular execution engines include Apache Spark [59], Dryad [39, 57], and Apache Tez [7]. Execution engines such as Tez [7] and DryadLINQ [57] allow for dynamic optimizations to the job DAG in the form of dynamism in vertex parallelism, data partitioning, and aggregation tree but lack the interfaces to make logical-level DAG switches.
Cluster Schedulers: Today's schedulers are multi-resource [30, 34, 43, 24, 17], DAG-aware [23, 34, 59], and allow a variety of constraints [61, 40, 18, 31, 58]. Given all these inputs, they optimize for objectives such as fairness [30, 41, 29, 20], performance [27], efficiency [34], or different combinations of the three [35, 36]. Over time, schedulers are becoming more complex and taking increasingly more job-level information as inputs. In contrast, we propose a simplified scheduler and argue for pushing complexity up the stack.

8 Conclusion

In this paper, we considered the problem of improving query performance in dynamic environments – e.g., in small private clusters, where resources vary with job arrivals and completions, and in clusters composed of spot instances, where resource availability changes due to changing prices. We showed that existing approaches are insufficient to adapt to dynamics because they use a fixed QEP throughout execution. We made the case for on-the-fly query re-planning and argued that it requires rethinking the division of labor among three key components of modern data analytics stacks: cluster scheduler, execution engine, and query planner. We propose a greedy re-planning algorithm, which offers provably competitive behavior, coupled with a simple cluster-wide scheduler that informs jobs of their current share. Our evaluation of a prototype using various workloads and resource profiles shows that our replanning approach driven by a simple scheduler matches or outperforms state-of-the-art solutions with complex schedulers and query planners.

Acknowledgements We thank the reviewers and our shepherd Angela Demke Brown. This work is supported by the National Science Foundation (grants CNS-1563011, CNS-1763810, CNS-1563095, CNS-1617773, and CCF-1617505), and Aditya Akella is also supported by a Google Faculty award, a gift from Huawei, and H. I. Romnes Faculty Fellowship.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[3] Apache Calcite. http://calcite.apache.org/.

[4] Apache Hadoop. http://hadoop.apache.org.

[5] Apache Hive. http://hive.apache.org.

[6] Apache Mesos 2016 Survey Report Highlights. https://goo.gl/R6a1z2.

[7] Apache Tez. http://tez.apache.org.

[8] Google Cluster Traces. https://github.com/google/cluster-data.

[9] Hadoop Private Cluster Size Statistics. https://wiki.apache.org/hadoop/PoweredBy.

[10] Presto. https://prestodb.io.

[11] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[12] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[13] Trident: Stateful stream processing on Storm. http://storm.apache.org/documentation/Trident-tutorial.html.

[14] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[15] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[16] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at Internet scale. *VLDB*, 2013.

[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[18] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[20] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[21] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[22] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[23] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[24] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[27] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[29] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[31] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[32] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[34] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[35] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[36] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[37] J. Hellerstein. Query optimization. In P. Bailis, J. M. Hellerstein, and M. Stonebraker, editors, *Readings in Database Systems*, chapter 7. 2017.

[38] B. Huang and J. Yang. CÜmÜlÖn-d: Data analytics in a dynamic spot market. *Proc. VLDB Endow.*, 10(8):865–876, Apr. 2017.

[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[40] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[41] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[42] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[43] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[44] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[45] H. Liu. Cutting MapReduce cost with spot market. In *HotCloud*, 2011.

[46] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[47] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic Query Re-planning using QOOP. Technical Report TR1855, University of Wisconsin-Madison, 2018.

[48] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[49] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[50] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataow system. In *SOSP*, 2013.

[51] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[52] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[53] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[54] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[55] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[56] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[57] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[58] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[59] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[60] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[61] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[62] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *SIGCOMM*, 2015.

Focus: Querying Large Video Datasets with Low Latency and Low Cost

Kevin Hsieh[†] Ganesh Ananthanarayanan[§] Peter Bodik[§] Shivaram Venkataraman[§ ✠]
Paramvir Bahl[§] Matthai Philipose[§] Phillip B. Gibbons[†] Onur Mutlu[⋆†]

[†]*Carnegie Mellon University* [§]*Microsoft* [✠]*University of Wisconsin* [⋆]*ETH Zürich*

Abstract

Large volumes of video are continuously recorded by cameras deployed for traffic control and surveillance with the goal of answering "after the fact" queries such as: *identify video frames with objects of certain classes (cars, bags)* from many days of recorded video. Current systems for processing such queries on large video datasets incur either high cost at video ingest time or high latency at query time. We present Focus, a system providing both low-cost and low-latency querying on large video datasets. Focus' architecture flexibly and effectively divides the query processing work between *ingest time* and *query time*. At ingest time (on live videos), Focus uses cheap convolutional network classifiers (CNNs) to construct an *approximate index* of all possible object classes in each frame (to handle queries for *any* class in the future). At query time, Focus leverages this approximate index to provide low latency, but compensates for the lower accuracy of the cheap CNNs through the judicious use of an expensive CNN. Experiments on commercial video streams show that Focus is 48× (up to 92×) cheaper than using expensive CNNs for ingestion, and provides 125× (up to 607×) lower query latency than a state-of-the-art video querying system (NoScope).

1. Introduction

Cameras are ubiquitous, with millions of them deployed by public and private entities at traffic intersections, enterprise offices, and retail stores. Videos from these cameras are continuously recorded [2, 6], with the main purpose of answering "after-the-fact" queries such as: *identify video frames with objects of certain classes (like cars or bags)* from many days of recorded video. Because the results from these video analytics queries may be needed quickly in many use cases, achieving low latency is crucial.

Advances in convolutional neural networks (CNNs) backed by copious training data and hardware accelerators (e.g., GPUs [12]) have led to highly accurate results in tasks like object detection and classification of images. For instance, the ResNet152 classifier CNN [45], winner of the ImageNet challenge 2015 [73], surpasses human-level performance in classifying 1,000 object classes on a public image dataset that has labeled ground truths [44].

Despite the accuracy of image classifier CNNs (like ResNet152) and object detectors (like YOLOv2 [68]), using them for video analytics queries is both expensive and slow. For example, even *after* using various motion detection techniques to filter out frames with no moving objects, using an object detector such as YOLOv2 [68] to identify frames with a given class (e.g., ambulance) on a month-long traffic video requires ≈ 190 hours on a high-end GPU (NVIDIA P100 [12]) and costs over $380 in the Azure cloud (Standard_NC6s_v2 instances). To achieve a query latency of say one minute on 190 GPU hours of work would require tens of thousands of GPUs detecting objects in the video frames in parallel, which is two to three orders of magnitude more than what is typically provisioned (few tens or hundreds of GPUs) by traffic jurisdictions or retail stores. Recent work like NoScope [51] has significantly improved the filtering of frames by using techniques like lightweight binary classifiers for the queried class (e.g., ambulance) before running heavy CNNs. However, the latencies are still long, e.g., it takes *5 hours* to query a month-long video on a GPU, in our evaluations. Moreover, videos from many cameras often need to be queried, which increases the latency and the GPU requirements even more.

The objective of our work is to enable *low-latency and low-cost querying over large historical video datasets*.

A natural approach to enable low latency queries is doing most of the work at *ingest-time*, i.e., on the *live* video that is being captured. If object detection, using say YOLO, were performed on frames at ingest-time, queries for specific classes (e.g., ambulance) would involve only a simple *index* lookup to find video frames with the queried object class. There are, however, two main shortcomings with this approach. First, most of the ingest-time work may be wasteful because typically only a small fraction of recorded frames ever get queried [16], e.g., only after an incident that needs investigation. Second, filtering techniques that use binary classifiers (as in NoScope [51]) are ineffective at ingest-time because *any* of a number of object classes could be queried later and running even lightweight binary classifiers for many classes can be prohibitively expensive.

Objectives & Techniques. We present Focus, a system to support low-latency, low-cost queries on large video datasets. To address the above challenges and shortcomings, Focus has the following goals: (a) provide low-cost indexing of *multiple* object classes in the video at ingest-time, (b) achieve high accuracy and low latency for queries, and (c) enable trade-offs between the cost at

ingest-time and the latency at query-time. Focus takes as inputs from the user a *ground-truth CNN* (or "GT-CNN", e.g., YOLO) and the desired accuracy of results that Focus needs to achieve relative to the GT-CNN. With these inputs, Focus uses three key techniques to achieve the above goals: (1) an *approximate* indexing scheme at ingest-time using cheap CNNs, (2) redundancy elimination by *clustering* similar objects, and (3) a tunable mechanism for judiciously *trading off* ingest cost and query latency.

(1) *Approximate indexing using a cheap ingest CNN.* To make video ingestion cheap, Focus uses *compressed* and *specialized* versions of the GT-CNN that have fewer convolutional layers [78], use smaller image sizes, and are trained to recognize the classes specific to each video stream. The cheap ingest CNNs, however, are less accurate than the expensive GT-CNN, both in terms of *recall* and *precision*. We define recall as the fraction of frames in the video that contain objects of the queried class that were *actually* returned in the query's results. Precision, on the other hand, is the fraction of frames in the query's results that contain objects of the queried class.

Using a cheap CNN to filter frames upfront risks incorrectly eliminating frames. To overcome this potential loss in recall, Focus relies on an empirical observation: while the top (i.e., most confident) classification results of the cheap CNNs and expensive GT-CNN often do not match, the top result of the expensive CNN often falls within the *top*-K most confident results of the cheap CNN. Therefore, at ingest-time, Focus indexes each frame with the "top-K" results of the cheap CNN, instead of just the top result. To increase precision, at query-time, after filtering frames using the top-K index, we apply the GT-CNN and return only frames that actually contains the queried object class.

(2) *Redundancy elimination via clustering.* To reduce the query-time latency of using the expensive GT-CNN, Focus relies on the significant similarity between objects in videos. For example, a car moving across a camera will look very similar in consecutive frames. Focus leverages this similarity by clustering the objects at ingest-time. We classify *only* the cluster centroids with the GT-CNN at query-time, and assign the same class to all objects in the cluster. This considerably reduces query latency. Clustering, in fact, identifies redundant objects even across non-contiguous and temporally-distant frames.

(3) *Trading off ingest cost vs. query latency.* Focus intelligently chooses its parameters (including K and the cheap ingest-time CNN) to meet user-specified targets on precision and recall. Among the parameter choices that meet the accuracy targets, it allows the user to trade off between ingest cost and query latency. For example, using a cheaper ingest CNN reduces the ingest cost but increases the query latency as Focus needs to use a larger K for the top-K index to achieve the accuracy targets. Focus automatically identifies the "sweet spot" in parameters, which sharply improves one of ingest cost or query latency for a small worsening of the other. It also allows for policies to balance the two, depending on the fraction of videos the application expects to get queried.

In summary, Focus' ingest-time and query-time operations are as follows. At ingest-time, Focus classifies the detected objects using a cheap CNN, clusters similar objects, and indexes each cluster centroid using the top-K most confident classification results, where K is auto-selected based on the user-specified precision, recall, and cost/latency trade-off point. At query-time, Focus looks up the ingest index for cluster centroids that match the class X requested by the user and classifies them using the GT-CNN. Finally, Focus returns all objects from the clusters that are classified as class X to the user.

Evaluation Highlights. We build Focus and evaluate it on fourteen 12-hour videos from three domains – traffic cameras, surveillance cameras, and news. We compare against two baselines: "Ingest-heavy", which uses the heavy GT-CNN for ingest, and "NoScope", a recent state-of-the-art video querying system [51]. We use YOLOv2 [68] as the GT-CNN. On average, across all the videos, Focus is 48× (up to 92×) cheaper than Ingest-heavy and 125× (up to 607×) faster than NoScope, all the while achieving ≥ 99% precision and recall. In other words, the latency to query a month-long video drops from 5 hours to only 2.4 minutes, at an ingest cost of $8/month/stream. Figure 1 also shows representative results with different trade-off alternatives for a surveillance video.

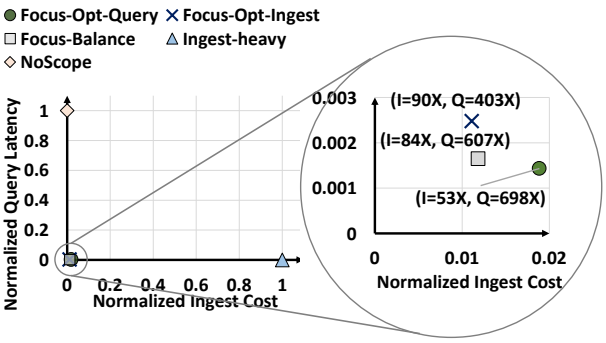


Figure 1: Effectiveness of Focus at reducing both ingest cost and query latency, for an example surveillance video. We compare against two baselines: "Ingest-heavy" that uses the YOLOv2 [68] object detector CNN for ingestion, and "NoScope", the state-of-the-art video querying system [51]. On the left, we see that Focus (the Focus-Balance point) is simultaneously 84× cheaper than Ingest-heavy in its cost (the I value) *and* 607× faster than NoScope in query latency (the Q value), all the while achieving at least 99% precision and recall (not plotted). Zooming in, also shown are two alternative Focus designs offering different trade-offs, Focus-Opt-Query and Focus-Opt-Ingest, each with at least 99% precision and recall.

Contributions: Our contributions are as follows.

- We present a new architecture for low-cost and low-latency querying over large video datasets, based on a principled split of ingest and query functionalities.
- We propose techniques for efficient indexing of multiple object classes: we create a top-K index at ingest time for high recall, while ensuring high precision by judiciously using expensive CNNs at query time.
- We show new policies that trade off between ingest cost and query latency: our system is significantly cheaper than an ingest-heavy design and significantly faster than query-optimized techniques like NoScope.

2. Background and Motivation

We first provide a brief overview of convolutional Neural Networks, the state-of-the-art approach to detecting and classifying objects in images (§2.1). We then discuss new observations we make about real-world videos, which motivate the design of our techniques (§2.2).

2.1. Convolutional Neural Networks

Convolution Neural Networks (CNNs) are the state-of-the-art method for many computer vision tasks such as object detection and classification (e.g., [45, 53, 59, 68, 84]).

Figure 2 illustrates the architecture of a representative image classification CNN. Broadly, CNNs consist of different types of layers including convolutional layers, pooling layers and fully-connected layers. The output from the final layer of a classification CNN is the probabilities of all object classes (e.g., dog, flower, car), and the class with the highest probability is the predicted class for the object in the input image.

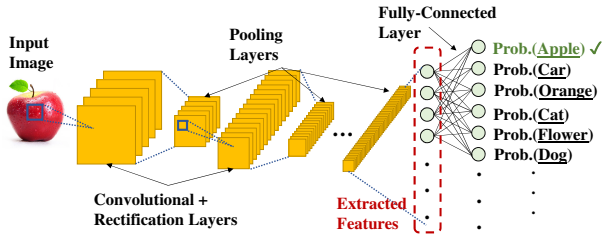


Figure 2: Architecture of an image classification CNN.

The output of the penultimate (i.e., previous-to-last) layer can be considered as "representative features" of the input image [53]. The features are a real-valued vector, with lengths between 512 and 4096 in state-of-the-art classifier CNNs (e.g., [45, 53, 78, 84]). It has been shown that images with similar feature vectors (i.e., small Euclidean distances) are visually similar [24, 53]. Thus, the distance between feature vectors is a standard metric to measure similarity of images in many applications, such as face recognition (e.g., [47]) and image retrieval (e.g., [23, 24, 67]).

Because *inference* using state-of-the-art CNNs is computationally expensive (and slow), two main techniques have been developed to reduce the cost of inference. First, *compression* is a set of techniques that can dramatically reduce the cost of inference at the expense of *accuracy*. Such techniques include removing some expensive convolutional layers [78], matrix pruning [34, 42], reducing input image resolution [68], and others [48, 71]. For example, ResNet18, which is a ResNet152 variant with only 18 layers, is 8× cheaper. Likewise, Tiny YOLO [68], a shallower variant of the YOLO object detector, is 5× cheaper than YOLOv2. However, the tradeoff is that compressed CNNs are usually less accurate than the original CNNs.

The second technique is CNN *specialization* [43], where the CNNs are trained on a subset of a dataset specific to a particular context (such as a video stream). Specialization *simplifies* the task of a CNN because specialized CNNs only need to consider a particular context. For example, differentiating object classes in any possible video is much more difficult than doing so in a traffic video, which is likely to contain far fewer object classes (e.g., cars, bicycles, pedestrians). As a result, specialized CNNs can be *more accurate* and *smaller* at the expense of generality. Leveraging compressed and specialized CNNs is a key facet of our solution (see §4).

2.2. Characterizing Real-world Videos

We aim to support queries of the form: *find all frames in the video that contain objects of class X*. We identify some key characteristics of real-world videos towards supporting these queries: (i) large portions of videos can be excluded (§2.2.1), (ii) only a limited set of object classes occur in each video (§2.2.2), and (iii) objects of the same class have similar feature vectors (§2.2.3). The design of Focus is based on these characteristics.

We analyze six 12-hour videos from three domains: traffic cameras, surveillance cameras, and news channels (§6.1 provides the details.) In this paper, we use results from YOLOv2 [68], trained to classify 80 object classes based on the COCO [60] dataset, as the ground truth.

2.2.1. Excluding large portions of videos. We find considerable potential to avoid processing large portions of videos at query-time. Not all the frames in a video are relevant to a query because each query looks only for a *specific class* of objects. In our video sets, an object class occurs in only 0.16% of the frames on average, and even the most frequent object classes occur in no more than $26\% - 78\%$ of the frames. This is because while there are usually some dominant classes (e.g., cars in a traffic camera, people in a news channel), most other classes are rare. Overall, the above data suggests considerable potential to speed up query latencies by *indexing* frames using the object classes. Also, in our experience, a system for querying videos is more useful for less frequent classes:

querying for "ambulance" in a traffic video is more interesting than querying for something commonplace like "car".

2.2.2. Limited set of object classes in each video. Most video streams have a limited set of objects because each video has its own context (e.g., traffic cameras can have automobiles, pedestrians or bikes, but not airplanes).

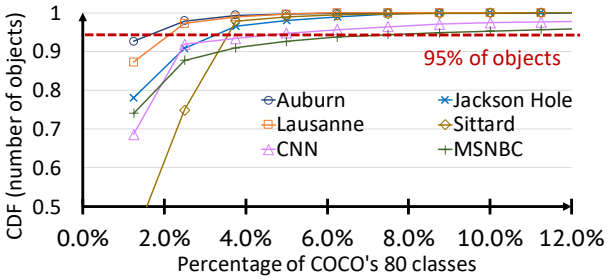


Figure 3: CDF of frequency of object classes. The x-axis is the fraction of classes out of the 80 classes recognized by the COCO [60] dataset (truncated to 12%).

Figure 3 shows the cumulative distribution function (CDF) of the frequency of object classes in our videos (as classified by YOLOv2). We make two observations. First, $2\% - 10\%$ of the most frequent object classes cover $\geq 95\%$ of the objects in all video streams. In fact, for some videos like `Auburn` and `Jackson Hole` we find that only $11\% - 19\%$ object classes occur in the entire video. Thus, for each video stream, if we can *automatically* determine its most frequent object classes, we can train efficient CNNs *specialized* for these classes. Second, a closer analysis reveals that there is little overlap between the object classes among different videos. On average, the Jaccard index [85] (i.e., intersection over union) between the videos based on their object classes is only 0.46. This implies that we need to specialize CNNs for each video stream separately to achieve the most benefits.

2.2.3. Feature vectors for finding duplicate objects. Objects moving in the video often stay in the frame for several seconds; for example, a pedestrian might take 15 seconds to cross a street. Instead of classifying *each instance* of the same object across the frames, we would like to *inexpensively* find duplicate objects and only classify one of them using a CNN (and apply the same label to all duplicates). Thus, given n duplicate objects, we would like only one CNN classification operation instead of n.

Comparing pixel values across frames is an obvious technique to identify duplicate objects, however, this technique turns out to be highly sensitive to even small changes in the camera's view of an object. Instead, feature vectors extracted from the CNNs (§2.1) are more robust because they are specifically trained to extract visual features for classification. We verify the robustness of feature vectors using the following analysis. In each video, for each object i, we find its *nearest* neighbor j using feature vectors from a cheap CNN (ResNet18) and compute the fraction of object pairs that belong to the same class. This fraction is over 99% in each of our videos, which shows the promise of using feature vectors from cheap CNNs to identify duplicate objects *even* across frames that are *not* temporally contiguous.

3. Overview of Focus

The goal of Focus is to *index live video streams* by the object classes occurring in them and enable answering "after-the-fact" queries later on the stored videos of the form: *find all frames that contain objects of class X*. Optionally, the query can be restricted to a subset of cameras and a time range. Such a query formulation is the basis for many widespread applications and could be used either on its own (such as for detecting all cars or bicycles in the video) or used as a basis for further processing (e.g., finding all collisions between cars and bicycles).

System Configuration. Focus is designed to work with a wide variety of current and future CNNs. The user (system administrator) provides a *ground-truth CNN* (GT-CNN), which serves as the accuracy baseline for Focus, but is far too costly to run on every video frame. Through a sequence of techniques, Focus provides results of nearly-comparable accuracy but at greatly reduced cost. In this paper, we use YOLOv2 [68] as the default GT-CNN.

Because different applications require different accuracies, Focus permits the user to specify the accuracy target, while providing reasonable defaults. The accuracy target is specified in terms of *precision*, i.e., fraction of frames output by the query that actually contain an object of class X according to GT-CNN, and *recall*, i.e., fraction of frames that contain objects of class X according to GT-CNN that were actually returned by the query.

Architecture: Figure 4 overviews the Focus design.

- At *ingest-time* (left part of Figure 4), Focus classifies objects in the incoming video frames and extracts their feature vectors. For its ingest, Focus uses highly compressed and specialized alternatives of the GT-CNN model (IT_1 in Figure 4). Focus then clusters objects based on their feature vectors (IT_2) and assigns to each cluster the *top* K most likely classes these objects belong to (based on classification confidence of the ingest CNN) (IT_3). It creates a *top*-K *index*, which maps each class to the set of object clusters (IT_4). The top-K index is the output of Focus' ingest-time processing of videos.
- At *query-time* (right part of Figure 4), when the user queries for a certain class X (QT_1), Focus retrieves the matching clusters from the top-K index (QT_2), runs the *centroids* of the clusters through GT-CNN (QT_3), and returns all frames from the clusters whose centroids were classified by GT-CNN as class X (QT_4).

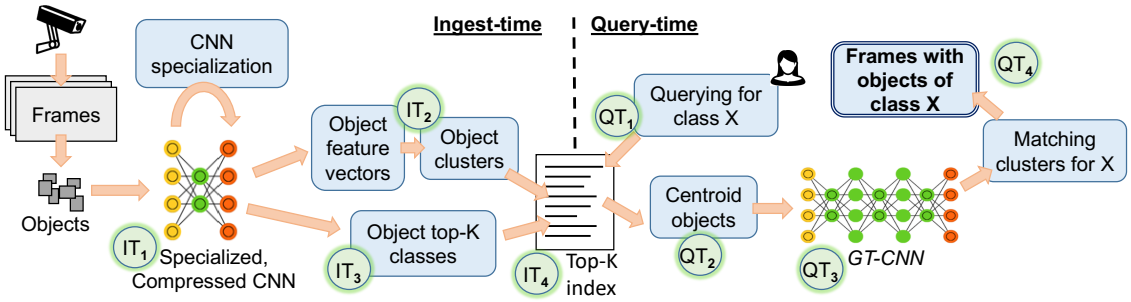


Figure 4: Overview of Focus.

The top-K ingest index is a mapping between the object classes and the clusters. In particular, we create a mapping from each object class to the clusters with top K matching object classes. Separately, we store the mapping between clusters and their corresponding objects and frames. The structure of the index is:

```
object class → ⟨cluster ID⟩
cluster ID → [centroid object, ⟨objects⟩ in
cluster, ⟨frame IDs⟩ of objects]
```

We next explain how Focus' key techniques keep ingest cost and query latency low while also meeting the user-specified recall and precision targets.

1) Top-K index via cheap ingesting: Focus makes indexing at ingest-time cheap by using compressed and specialized alternatives of the GT-CNN for *each* video stream. *Compression* of CNNs [34, 42, 48, 78] uses fewer convolutional layers and other approximations (§2.1), while *specialization* of CNNs [43,75] uses the observation that a specific video stream contains only a small number of object classes and their appearance is more constrained than in a generic video (§2.2.2). Both optimizations are done automatically by Focus and together result in ingest-time CNNs that are up to 96× cheaper than the GT-CNN.

The cheap ingest-time CNNs are less accurate, i.e., their top-most results often do not match the top-most classifications of GT-CNN. Therefore, to improve recall, Focus associates each object with the *top*-K classification results of the cheap CNN, instead of only its top-most result. Increasing K increases recall because the top-most result of GT-CNN often falls within the ingest-time CNN's top-K results. At query-time, Focus uses the GT-CNN to remove objects in this larger set that do not match the class, to regain the precision lost by including the top-K.

2) Clustering similar objects. A high value of K at ingest-time increases the work done at query time, thereby increasing query latency. To reduce this overhead, Focus clusters similar objects at ingest-time using feature vectors from the cheap ingest-time CNN (§2.2.3). In each cluster, at query-time, we run only the cluster centroid through GT-CNN and apply the classified result from the GT-CNN to all objects in the cluster. Thus, a tight clustering of objects is crucial for high precision and recall.

3) Trading off ingest vs. query costs. Focus automatically chooses the ingest CNN, its K, and specialization and clustering parameters to achieve the desired precision and recall targets. These choices also help Focus trade off between the work done at ingest-time and query-time. For instance, to save ingest work, Focus can select a cheaper ingest-time CNN, and then counteract the resultant loss in recall by using a higher K and running the expensive GT-CNN on more objects at query time. Focus chooses its parameters so as to offer a sharp improvement in one of the two costs for a small degradation in the other cost. Because the desired trade-off point is application-dependent, Focus provides users with options: "ingest-optimized", "query-optimized", and "balanced" (the default). Figure 1 (§1) presents an example result.

4. Video Ingest & Querying Techniques

We describe the main techniques used in Focus: constructing approximate indexes with cheap CNNs at ingest-time (§4.1), specializing the CNNs to the specific videos (§4.2), and identifying similar objects and frames to save on redundant CNN processing (§4.3). §4.4 describes how Focus flexibly trades off ingest cost and query latency.

4.1. Approximate Index via Cheap Ingest

Focus indexes the live videos at *ingest-time* to reduce the *query-time* latency. We detect and classify the objects within the frames of the live videos using *ingest-time* CNNs that are far cheaper than the ground-truth GT-CNN. We use these classifications to index objects by class.

Cheap ingest-time CNN. As noted earlier, the user provides Focus with a GT-CNN. Optionally, the user can also provide other CNN architectures to be used in Focus' search for cheap CNNs. Examples include object detector CNNs (which vary in their resource costs and accuracies) like YOLO [68] and Faster RCNN [69] that jointly detect the objects in a frame and classify them. Alternatively, objects can be detected separately using relatively inexpensive techniques like background subtraction [28], which are well-suited for static cameras, as in surveillance or traffic installations, and then the detected objects can be classified using object classification CNN architectures

such as ResNet [45], AlexNet [53] and VGG [78].[1]

Starting from these user-provided CNNs, Focus applies various levels of compression, such as removing convolutional layers and reducing the input image resolution (§2.1). This results in a large set of CNN options for ingest, $\{\text{CheapCNN}_1, \ldots, \text{CheapCNN}_n\}$, with a wide range of costs and accuracies, out of which Focus picks its ingest-time CNN, $\text{CheapCNN}_{\text{ingest}}$.

Top-K Ingest Index. To keep recall high, Focus indexes each object using the *top* K object classes from the output of $\text{CheapCNN}_{\text{ingest}}$, instead of using just the top-most class. Recall from §2.1 that the output of the CNN is a list of classes for each object in descending order of confidence. We make the following *empirical* observation: the top-most output of the expensive GT-CNN for an object is often contained within the top-K classes output by the cheap CNN, for a small value of K.

Figure 5 demonstrates the above observation by plotting the effect of K on recall on one of our video streams from a static camera, `lausanne` (see §6.1). We explore many cheaper ResNet18 [45] models by removing one layer at a time with various input image sizes. The trend is the same among the CNNs we explore so we present three models for clarity: ResNet18, and ResNet18 with 4 and 6 layers removed; correspondingly to each model, the input images were rescaled to 224, 112, and 56 pixels, respectively. These models were also *specialized* to the video stream (more in §4.2). We make two observations.

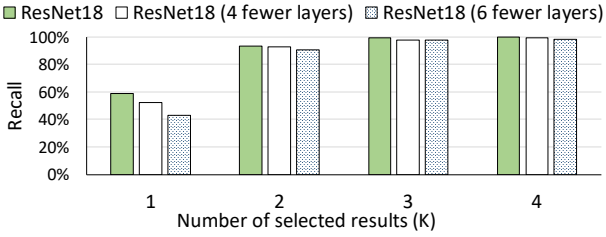


Figure 5: Effect of K on the recall of three cheap classifier CNNs to classify the detected objects. Recall is measured relative to the results of the GT-CNN, YOLOv2 [68].

First, we observe steady increase in recall with increasing K, for all three CheapCNNs. As the figure shows, all the cheap CNNs reach $\geq 99\%$ recall when $K \geq 4$. Note that all these models recognize 80 classes, so $K = 4$ represents only 5% of the possible classes. Second, there is a *trade-off* between different models – typically, the cheaper they are, the lower their recall with the same K. However, we can compensate for the loss in recall in cheaper models using a larger K to reach a certain recall value. Overall, we conclude that by selecting the appropriate model and K, Focus can achieve the target recall.

Achieving precision. Focus creates the *top*-K *index* from the top-K classes output by $\text{CheapCNN}_{\text{ingest}}$ for every object at ingest-time. While filtering for objects of the queried class X using the top-K index (with the appropriate K) will have a high recall, this will lead to very low precision. Because we associate each object with K classes (while it has only one true class), the average precision is only $1/K$. Thus, at query time, to improve precision, Focus determines the *actual* class of objects from the top-K index using the expensive GT-CNN and returns only the objects that match the queried class X.

Skipping GT-CNN for high-confidence indexes. Focus records the prediction confidence along with the top-K results by $\text{CheapCNN}_{\text{ingest}}$. The system can skip invoking GT-CNN for the indexes with prediction confidence higher than a chosen threshold (Skip_{th}). Not invoking GT-CNN for these indexes can cause precision to fall if the threshold is too low. Hence, this parameter needs to be carefully selected to retain high precision.

Parameter selection. The selection of the cheap ingest-time CNN model ($\text{CheapCNN}_{\text{ingest}}$) and the K value (for the top-K results) has a significant influence on the recall of the output produced. Lower values of K reduce recall, i.e., Focus will miss frames that contain the queried objects. At the same time, higher values of K increase the number of objects to classify with GT-CNN at query time, and hence adds to the latency. §4.4 describes how Focus sets these parameters because they have to be jointly set with other parameters described in §4.2 and §4.3.

4.2. Video-specific Specialization of Ingest CNN

To further reduce the ingest cost, Focus *specializes* the ingest-time CNN model to each video stream. As §2.1 describes, model specialization [43] reduces cost by simplifying the task of CNNs. Specifically, model specialization takes advantage of two characteristics in real-world videos. First, most video streams have a limited set of object classes (§2.2.2). Second, objects in a specific stream are often *visually more constrained* than objects in general (say, in the COCO [60] dataset). The cars and buses that occur in a specific traffic camera have much less variability, e.g., they have very similar angle, distortion and size, compared to a generic set of vehicle images. Thus, classifying objects from a specific camera is a much simpler task than doing so from all cameras, resulting in cheaper ingest-time CNNs.

While specializing CNNs to specific videos has been attempted in computer vision research (e.g., [43, 75]), we explain its two key implications within Focus.

1) Lower K values. Because the specialized CNN classifies across fewer classes, they are more accurate, which enables Focus to achieved the desired recall with a much smaller K (for the top-K ingest index). We find that spe-

[1] Focus is agnostic to whether object detection and classification are done together or separately. In practice, the set of detected object bounding boxes (but not their classifications!) remain largely the same with different ingest CNNs, background subtraction, and the GT-CNN.

cialized models can usually use $K \leq 4$ (Figure 5), much smaller compared to the typical K needed for *generic* cheap CNNs. A smaller K translates to fewer objects that have to be classified by GT-CNN at query time, thus reducing latency.

2) Most frequent classes. On each video stream, Focus periodically obtains a small sample of video frames and classifies their objects using GT-CNN to estimate the ground truth of the distribution of object classes for the video (similar to Figure 3). From this distribution, Focus selects the most frequently occurring L_s object classes to retrain new specialized models. Because just a handful of classes often account for a dominant majority of the objects (§2.2.2), low values of L_s usually suffice.

While Focus specializes the CNN towards the most frequently occurring L_s classes, we also want to support querying of the *less* frequent classes. For this purpose, Focus includes an additional class called "OTHER" in the specialized model. Being classified as OTHER simply means not being one of the L_s classes. At query time, if the queried class is among the OTHER classes of the ingest CNN's index, Focus extracts all the clusters that match the OTHER class and classifies their centroids through the GT-CNN model.[2]

The parameter L_s (for each video stream) exposes the following trade-off. Using a small L_s enables us to train a simpler model with cheaper ingest cost and lower query-time latency *for the popular classes*, but, it also leads to a larger fraction of objects falling in the OTHER class. As a result, querying for the OTHER class will be expensive because all those objects will have to be classified by the GT-CNN. Using a larger value of L_s, on the other hand, leads to more expensive ingest and query-time models, but cheaper querying for the OTHER classes. We select L_s in §4.4.

4.3. Redundant Object Elimination

At query time, Focus retrieves the objects likely matching the user-specified class from the top-K index and infers their actual class using the GT-CNN. This ensures precision of 100%, but could cause significant latency at query time. Even if this inference were parallelized across many GPUs, it would incur a large cost. Focus uses the following observation to reduce this cost: if two objects are visually similar, their feature vectors are also similar and they would likely be classified as the same class (e.g., cars) by the GT-CNN model (§2.2.3).

Focus *clusters* objects that are similar, invokes the expensive GT-CNN only on the cluster centroids, and assigns the centroid's label to all objects in each cluster. Doing so dramatically reduces the work done by the GT-CNN classifier at query time. Focus uses the feature vector output by the previous-to-last layer of the cheap ingest CNN (see §2.1) for clustering. Note that Focus clusters the *objects* in the frames and not the frames as a whole.[3]

The key questions regarding clustering are *how* we cluster and *when* we cluster. We discuss both below.

Clustering Heuristic. We require two properties in our clustering technique. First, given the high volume of video data, it should be a single-pass algorithm to keep the overhead low, unlike most clustering algorithms, which are *quadratic* complexity. Second, it should make no assumption on the number of clusters and adapt to outliers in data points on the fly. Given these requirements, we use the following simple approach for *incremental* clustering, which has been well-studied in the literature [30, 65].

We put the first object into the first cluster c_1. To cluster a new object i with a feature vector f_i, we assign it to the closest cluster c_j if c_j is at most distance T away from f_i, where T is a distance threshold. However, if none of the clusters are within a distance T, we create a new cluster with centroid at f_i. We measure distance as the L_2 norm [9] between the cluster centroid feature vector and the object feature vector f_i. To bound the time complexity for clustering, we keep the number of clusters actively being updated at a constant C. We do this by *sealing* the smallest cluster when the number of clusters hits $C+1$, but we keep growing the popular clusters (such as similar cars). This maintains the complexity as $O(Cn)$, which is linear in n, the total number of objects. The value of C has a very minor impact on our evaluation results, and we set C as 100 in our evaluations.

Clustering can reduce precision and recall depending on the parameter T. If the centroid is classified by GT-CNN as the queried class X but the cluster contains another object class, it reduces precision. If the centroid is classified as a class different than X but the cluster has an object of class X, it reduces recall. §4.4 discuss setting T.

Clustering at Ingest vs. Query Time. Focus clusters the objects at ingest-time rather than at query-time. Clustering at query-time would involve *storing* all feature vectors, *loading* them for objects filtered from the ingest index and then clustering them. Instead, clustering *at ingest time* creates clusters right when the feature vectors are created and stores only the cluster centroids in the top-K index. This makes the query-time latency much lower and also reduces the size of the top-K index. We observe that the ordering of indexing and clustering operations is mostly *commutative* in practice and has little impact

[2] Specialized CNNs can be retrained quickly on a small dataset. Retraining is relatively infrequent and done once every few days. Also, because there will be considerably fewer objects in the video belonging to the OTHER class, we proportionally re-weight the training data to contain equal number of objects of all the classes.

[3] Recall from §4.1 that Focus' ingest process either (i) employs an object detector CNN (e.g., YOLO) that jointly detects and classifies objects in a frame; or (ii) detects objects with background subtraction and then classifies objects with a classifier CNN (e.g. ResNet). Regardless, we obtain the feature vector from the CNNs for *each object* in the frame.

on recall and precision (we do not present these results due to space constraints). We therefore use ingest-time clustering due to its latency and storage benefits.

4.4. Trading off Ingest Cost and Query Latency

Focus' goals of high recall/precision, low ingest cost and low query latency are affected by its parameters: (i) K, the number of top results from the ingest-time CNN to index an object; (ii) L_s, the number of popular object classes we use to create a specialized model; (iii) CheapCNN$_i$, the specialized ingest-time cheap CNN; (iv) Skip$_{\text{th}}$, the confidence threshold to skip invoking GT-CNN; and (v) T, the distance threshold for clustering objects.

Viable Parameter Choices. Focus first prunes the parameter choices to only those that meet the desired precision and recall targets. Among the five parameters, four parameters (K, L_s, CheapCNN$_i$, and T) impact recall; only T and Skip$_{\text{th}}$ impact precision. Focus samples a representative fraction of the video stream and classifies them using GT-CNN for the ground truth. Next, for each combination of parameter values, Focus computes the precision and recall (relative to GT-CNN's outputs) achievable for each of the object classes, and selects only those combinations that meet the precision and recall targets.

Among the viable parameter choices that meet the precision and recall targets, Focus *balances* ingest- and query-time costs. For example, picking a more accurate CheapCNN$_{\text{ingest}}$ will have higher ingest cost, but lower query cost because we can use a smaller K. Using a less accurate CheapCNN$_{\text{ingest}}$ will have the opposite effect.

Pareto Boundary. Focus identifies "intelligent defaults" that sharply improve one of the two costs for a small worsening of the other cost. Figure 6 illustrates the tradeoff between ingest cost and query latency for one of our video streams. The figure plots all the viable "configurations" (i.e., parameter choices that meet the precision and recall targets) based on their ingest cost (i.e., cost of CheapCNN$_{\text{ingest}}$) and query latency (i.e., the number of clusters that need to be checked at query time according to K, L_s, T and Skip$_{\text{th}}$).

We first extract the *Pareto boundary* [17], which is defined as the set of configurations among which we cannot improve one of the metrics without worsening the other. For example, in Figure 6, the yellow triangles are not Pareto optimal when compared to the points on the dashed line. Focus can discard all non-Pareto configurations because at least one point on the Pareto boundary is better than all non-Pareto points in *both* metrics.

Tradeoff Policies. Focus balances ingest cost and query latency (Balance in Figure 6) by selecting the configuration that minimizes the *sum of ingest cost and query latency*. We measure ingest cost as the compute cycles taken to ingest the video and query latency as the average time (or cycles) required to query the video on the object

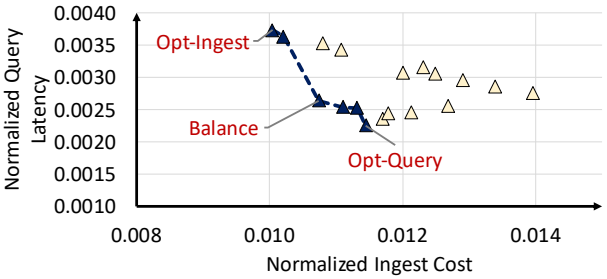


Figure 6: Parameter selection based on the ingest cost and query latency trade-off. The ingest cost is normalized to the cost of ingesting all video frames with GT-CNN (YOLOv2), while the query latency is normalized to the query latency using NoScope. The dashed line is the Pareto boundary.

classes that are recognizable by the ingest CNN. By default, Focus chooses a Balance policy that equally weighs ingest cost and query latency. Users can also provide any other weighted function to optimize their goal.

Focus also allows for other configurations based on the application's preferences and query rates. Opt-Ingest minimizes the ingest cost and is applicable when the application expects most of the video streams to not get queried (such as surveillance cameras), as this policy minimizes the amount of wasted ingest work. On the other hand, Opt-Query minimizes query latency but it incurs a larger ingest cost. More complex policies can be easily implemented by changing how the query latency cost and ingest cost are weighted in our cost function. Such flexibility enables Focus to fit a number of applications.

5. Implementation

Because Focus targets large video datasets, a key requirement of Focus' implementation is the ability to scale and distribute computation across many machines. To this end, we implement Focus as three loosely-coupled modules which handle each of its three key tasks. Figure 7 presents the architecture and the three key modules of Focus: the *ingest processor* (M1), the *stream tuner* (M2), and the *query processor* (M3). These modules can be flexibly deployed on different machines based on the video dataset size and the available hardware resources (such as GPUs). We describe each module in turn.

5.1. Ingest Processor

Focus' ingest processor (M1) generates the approximate index (§4.1) for the input video stream. The work is distributed across many machines, with each machine running one worker process for *each* video stream's ingestion. An ingest processor handles its input video stream with a four-stage pipeline: (i) extracting the moving objects from the video frames (IP$_1$ in Figure 7), (ii) inferring the top-K indexes and the feature vectors of all detected objects with the ingest-time CNN (IP$_2$ in Figure 7, §4.1),

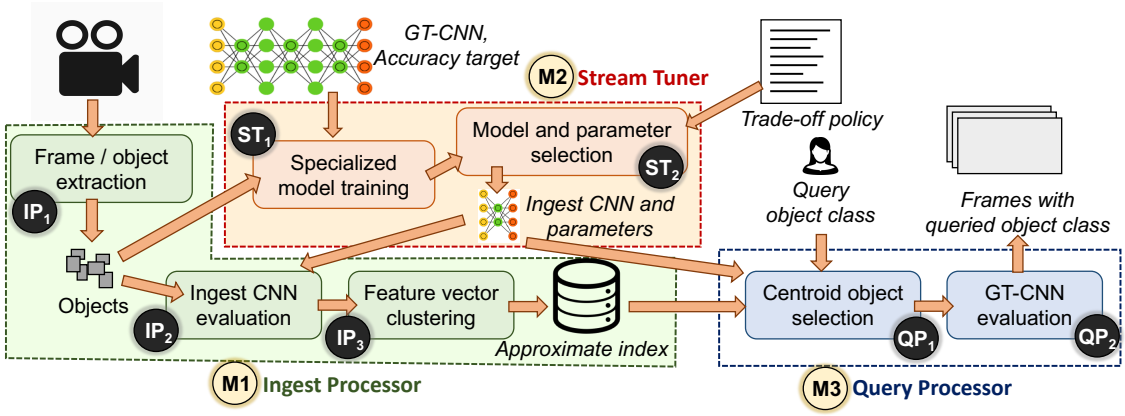


Figure 7: Key components of Focus.

(iii) using the feature vector to cluster objects (IP_3 in Figure 7, §4.3), and (iv) storing the top-K indexes of centroid objects in a database for efficient retrieval at query time.

An ingest processor is configured differently for static (fixed-angle) and moving cameras. For static cameras, we extract object boxes by subtracting each video frame from the *background frame*, which is obtained by averaging the frames in each hour of the video. We then index each object box with an ingest-time *object classifier* CNN. We accelerate the background subtraction with GPUs [14]. We use background subtraction for static cameras because running background subtraction with a cheap object classifier is much faster than running an ingest-time *object detector* CNN, and we find that both approaches have almost the same accuracy in detecting objects in static cameras. Hence, we choose the cheaper ingest option.

For moving cameras, we use a cheap, ingest-time *object detector* CNN (e.g., Tiny YOLO [68]) to generate the approximate indexes. We choose the object detection threshold (the threshold to determine if a box has an object) for the object detector CNN such that we do not miss objects in GT-CNN while minimizing spurious objects.

5.2. Stream Tuner

The stream tuner (M2) determines the ingest-time CNN and Focus' parameters for each video stream (§4.4). It takes four inputs: the sampled frames/objects, the GT-CNN, the desired accuracy relative to the GT-CNN, and the tradeoff policy between ingest cost and query latency (§4.4). Whenever executed, the stream tuner: (i) generates the ground truth of the sampled frames/objects with the GT-CNN; (ii) trains specialized ingest-time CNNs based on the ground truth (ST_1 in Figure 7); and (iii) selects the ingest-time CNN and Focus' parameters (ST_2 in Figure 7).

Focus executes the stream tuner for each video stream *before* launching the corresponding ingest processor. As the characteristics of video streams may change over time, Focus periodically launches the stream tuner to validate the accuracy of the selected parameters on sampled frames. The ingest-time CNN and the system parameters are re-tuned if necessary to meet the accuracy targets.

5.3. Query Processor

The task of the query processor is to return the video frames that contain the user's queried object class. In response to a user query for class X, the query processor first retrieves the centroid objects with matching approximate indexes (QP_1 in Figure 7), and then uses the GT-CNN to determine the frames that do contain object class X (QP_2 in Figure 7, §4.1). The GT-CNN evaluation can be easily distributed across many machines, if needed.

We employ two optimizations to reduce the overhead of GT-CNN evaluation. First, we skip the GT-CNN evaluation for high-confidence indexes (§4.1). Second, we apply a query-specialized binary classifier [51] on the frames that need to be checked before invoking the GT-CNN. These two optimizations make the query processor more efficient by *not* running GT-CNN on all candidate centroid objects.

6. Evaluation

We evaluate our Focus prototype with more than 160 hours of videos from 14 real video streams that span traffic cameras, surveillance cameras, and news channels. Our main results are:

- Focus is simultaneously 48× cheaper on average (up to 92×) than the Ingest-heavy baseline in processing videos and 125× faster on average (up to 607×) than NoScope [51] in query latency — all the while achieving at least 99% precision and recall (§6.2, §6.3).
- Focus provides a rich trade-off space between ingest cost and query latency. If a user wants to optimize for *ingest cost*, Focus is 65× cheaper on average (up to 96×) than the Ingest-heavy baseline, while reducing query latency by 100× on average. If the goal is to optimize for *query latency*, Focus can achieve 202× (up to 698×) faster queries than NoScope with 53× cheaper ingest. (§6.4).

Table 1: Video dataset characteristics

Type	Camera	Name	Description
Traffic	Static	auburn_c	A commercial area intersection in the City of Auburn [5]
		auburn_r	A residential area intersection in the City of Auburn [4]
		bellevue_d	A downtown intersection in the City of Bellevue. The video streams are obtained from city traffic cameras.
		bellevue_r	A residential area intersection in the City of Bellevue
		bend	A road-side camera in the City of Bend [7]
		jackson_h	A busy intersection in Jackson Hole [8]
		jackson_ts	A night street in Jackson Hole. The video is downloaded from the NoScope project website [50].
Surveillance	Static	coral	An aquarium video downloaded from the NoScope project website [50]
		lausanne	A pedestrian plaza (Place de la Palud) in Lausanne [10]
		oxford	A bookshop street in the University of Oxford [15]
		sittard	A market square in Sittard [3]
News	Moving	cnn	News channel
		foxnews	News channel
		msnbc	News channel

6.1. Methodology

Software Tools. We use OpenCV 3.4.0 [13] to decode the videos into frames, and we feed the frames to our evaluated systems, Focus and NoScope. Focus runs and trains CNNs with Microsoft Cognitive Toolkit 2.4 [64], an open-source deep learning system. Our ingest processor (§5.1) stores the approximate index in MongoDB [11] for efficient retrieval at query time.

Video Datasets. We evaluate 14 video streams that span across traffic cameras, surveillance cameras, and news channels. We record each video stream for 12 hours to cover both day time and night time. Table 1 summarizes the video characteristics. We strengthen our evaluation by including down sampling (or frame skipping), one of the most straightforward approaches to reduce ingest cost and query latency, into our evaluation *baseline*. Specifically, as the vast majority of objects show up for at least one second in our evaluated videos, we evaluate each video at 1 fps instead of 30 fps. We find that the object detection results at these two frame rates are almost the same. Each video is split evenly into a *training set* and a *test set*. The training set is used to train video-specialized CNNs and select system parameters. We then evaluate the systems with the test set. In some figures, we show results for only eight representative videos to improve legibility.

Accuracy Target. We use YOLOv2 [68], a state-of-the-art object detector CNN, as our ground-truth CNN (GT-CNN): all objects detected by GT-CNN are considered to be the correct answers.[4] For each query, our default accuracy target is 99% recall and precision. To avoid over-fitting, we use the *training set* of each video to explore system parameters with various recall/precision targets (i.e., 100%–95% with a 0.5% step), and we report the best system parameters that can *actually* achieve the recall/precision target on the *test set*. We also evaluate other recall/precision targets such as 97% and 95% (§6.5).

Baselines. We use baselines at two ends of the design spectrum: (1) Ingest-heavy, the baseline system that uses GT-CNN to analyze all frames at ingest time, and stores the results as an index for query; and (2) NoScope, a recent state-of-the-art querying system [51] that analyzes frames for the queried object class at query time. We also use a third baseline, Ingest-NoScope that uses NoScope's techniques at ingest time. Specifically, Ingest-NoScope runs the binary classifiers of NoScope for all possible classes at *ingest time*, invokes GT-CNN if any of the binary classifiers cannot produce a high-confidence result, and stores the results as an index for query. To further strengthen the baselines, we augment all baseline systems with background subtraction, thus eliminating frames with no motion. As Focus is in the middle of the design spectrum, we compare Focus' ingest cost with Ingest-heavy and Ingest-NoScope, and we compare Focus' query latency with NoScope.

Metrics. We use two performance metrics. The first metric is *ingest cost*, the end-to-end machine time to ingest each video. The second metric is *query latency*, the end-to-end latency for an object class query. Specifically, for each video stream, we evaluate the object classes that collectively make up 95% of the detected objects in GT-CNN. We report the average query latency on these object classes. We do not evaluate the bottom 5% classes because they are often random erroneous results in GT-CNN (e.g., "broccoli" or "orange" in a traffic camera).

Both metrics include the time spent on all processing stages, such as detecting objects with background subtraction, running CNNs, clustering, reading and writing to the approximate index, etc. Similar to prior work [51, 68], we report the end-to-end execution time of each system while excluding the video decoding time, as the decoding time can be easily accelerated with GPUs or accelerators.

Experimental Platform. We run the experiments on Standard_NC6s_v2 instances on the Azure cloud. Each instance is equipped with a high-end GPU (NVIDIA Tesla P100), 6-core Intel Xeon CPU (E5-2690), 112 GB RAM, a 10 GbE NIC, and runs 64-bit Ubuntu 16.04 LTS.

6.2. End-to-End Performance

Static Cameras. We first show the end-to-end performance of Focus on static cameras when Focus aims to balance these two metrics (§4.4). Figure 8 compares the ingest cost of Focus and Ingest-NoScope with Ingest-heavy and the query latency of Focus with NoScope. We make three main observations.

[4] We do not use the latest YOLOv3 or other object detector CNN such as FPN [59] as our GT-CNN because one of our baseline systems, NoScope, comes with the YOLOv2 code. Fundamentally, there is no restriction on the selection of GT-CNN for Focus.

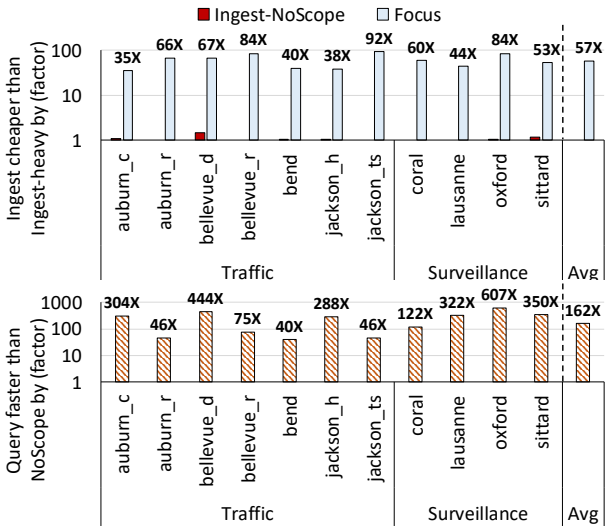


Figure 8: (Top) Focus **ingest cost compared to Ingest-heavy. (Bottom)** Focus **query latency compared to NoScope.**

First, Focus significantly improves query latency with a very small cost at ingest time. Focus achieves 162× speedup (on average) in query latency over NoScope with a very small ingest cost (57× cheaper than Ingest-heavy, on average), all the while retaining 99% recall and precision (not shown). Focus achieves two orders of magnitude speedup over NoScope because: (i) the ingest-time approximate indexing drastically narrows down the frames that need to be checked at query time; and (ii) the feature-based clustering further reduces the redundant work. In contrast, NoScope needs to go through *all* the frames at query time, which is especially inefficient for the object classes that appear infrequently. We conclude that Focus' architecture provides a valuable trade-off between ingest cost and query latency.

Second, directly applying NoScope's techniques at ingest time (Ingest-NoScope) does not save much cost over Ingest-heavy. There are two reasons for this: (1) While each binary classifier is relatively cheap, running multiple instances of binary classifiers (for all possible object classes) imposes non-trivial cost. (2) The system needs to invoke GT-CNN when any one of the binary classifiers cannot derive the correct answer. As a result, GT-CNN is invoked for most frames. Hence, the ingest cost of Focus is much cheaper than both, Ingest-heavy and Ingest-NoScope. This is because Focus' architecture only needs to construct the approximate index at ingest time which can be done cheaply with an ingest-time CNN.

Third, Focus is effective across videos with varying characteristics. It makes queries 46× to 622× faster than NoScope with a very small ingest cost (35× to 92× cheaper than Ingest-heavy) among busy intersections (`auburn_c`, `bellevue_d` and `jackson_h`), normal intersections (`auburn_r`, `bellevue_r`, `bend`), a night street (`jackson_ts`), busy plazas (`lausanne` and `sittard`), a university street (`oxford`), and an aquarium (`coral`). The gains in query latency are smaller for some videos (`auburn_r`, `bellevue_r`, `bend`, and `jackson_ts`). This is because Focus' ingest CNN is less accurate on these videos, and Focus selects more conservative parameters (e.g., a larger K such as 4–5 instead of 1–2) to attain the recall/precision targets. As a result, there is more work at query time for these videos. Nonetheless, Focus still achieves at least 40× speedup over NoScope in query latency. We conclude that the core techniques of Focus are general and effective on a variety of real-world videos.

Moving Cameras. We evaluate the applicability of Focus on moving cameras using three news channel video streams. These news videos were recorded with moving cameras and they change scenes between different news segments. For moving cameras, we use a cheap object detector (Tiny YOLO, which is 5× faster than YOLOv2 for the same input image size) as our ingest-time CNN. Figure 9 shows the end-to-end performance of Focus on moving cameras.

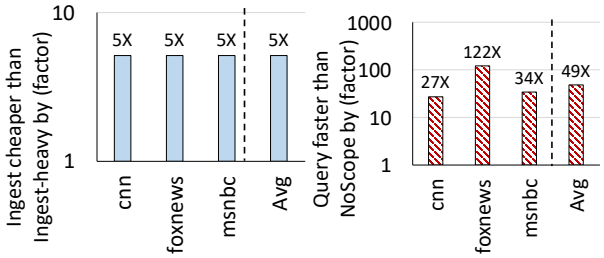


Figure 9: Focus **performance on moving cameras. (Left)** Focus **ingest cost compared to Ingest-heavy. (Right)** Focus **query latency compared to NoScope.**

As the figure shows, Focus is effective in reducing query latency with only a modest ingest cost. Focus achieves a 49× speedup in query latency on average over NoScope, with ingest cost that is 5× cheaper than Ingest-heavy. We make two main observations. First, the ingest cost improvements on moving cameras (5×) is lower than the ones on static cameras (57×). This is because moving cameras require a detector CNN to detect objects, and it is more costly to run a cheap object detector (like Tiny YOLO) as opposed to using background subtraction to detect the objects and then classifying them using a cheap classifier CNN (like compressed ResNet18). Our design, however, does not preclude using much cheaper detectors than Tiny YOLO, and we can further reduce the ingest cost of moving cameras by exploring even cheaper object detector CNNs. Second, Focus' techniques are very effective in reducing query latency on moving cameras. The approximate index generated by a cheap detector CNN significantly narrows down the frames that need to be

checked at query time. We conclude that the techniques of Focus are general and can be applied to a wide range of object detection CNNs and camera types.

Averaging over both static and moving cameras, Focus' ingest cost is 48× cheaper than Ingest-heavy and its queries are 125× faster than NoScope.

We now take a deeper look at Focus' performance using representative static cameras.

6.3. Effect of Different Focus Components

Figure 10 shows the breakdown of query latency gains for two core techniques of Focus: (1) Approximate indexing, which indexes each object with the top-K results of the ingest-time CNN, and (2) Approximate indexing + Clustering, which adds feature-based clustering at ingest time to reduce redundant work at query time. We show the results that achieve at least 99% recall and precision. We make two observations.

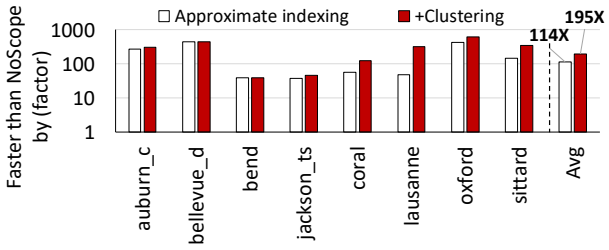


Figure 10: Effect of different Focus components on query latency reduction

First, approximate indexing is the major source of query latency improvement. This is because approximate indexing effectively eliminates irrelevant objects for each query and bypasses the query-time verification for high-confidence ingest predictions. As a result, only a small fraction of frames need to be resolved at query time. On average, approximate indexing alone is 114× faster than NoScope in query latency.

Second, clustering is a very effective technique to further reduce query latency. Using clustering (on top of approximate indexing) reduces the query latency by 195×, significantly better than approximate indexing alone. We see that clustering is especially effective on surveillance videos (e.g., `coral`, `lausanne`, and `oxford`) because objects in these videos tend to stay longer in the camera (e.g., "person" on a plaza compared to "car" in traffic videos), and hence there is more redundancy in these videos. This gain comes with a negligible cost because we run our clustering algorithm (§4.3) on the otherwise idle CPUs of the ingest machine while the GPUs run the ingest-time CNN model.

6.4. Ingest Cost vs. Query Latency Trade-off

One of the important features of Focus is the flexibility to tune its system parameters to achieve different application goals (§4.4). Figure 11 (the zoom-in region of Figure 1) depicts three alternative settings for Focus that illustrate the trade-off space between ingest cost and query latency, using the `oxford` video stream: (1) Focus-Opt-Query, which optimizes for query latency by increasing ingest cost, (2) Focus-Balance, which is the default option that balances these two metrics (§4.4), and (3): Focus-Opt-Ingest, which is the opposite of Focus-Opt-Query. The results are shown relative to the Ingest-heavy and NoScope baselines. Each data label (I, Q) indicates its ingest cost is $I\times$ cheaper than Ingest-heavy, while its query latency is $Q\times$ faster than NoScope.

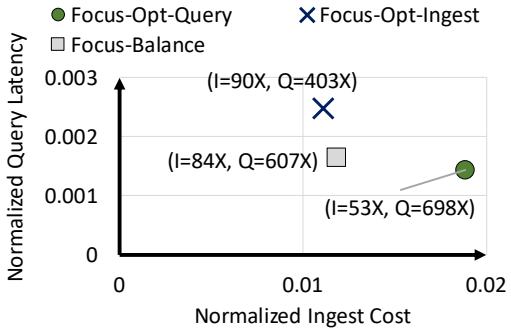


Figure 11: Focus' trade-off policies on an example video

As Figure 11 shows, Focus offers very good options in the trade-off space between ingest cost and query latency. Focus-Opt-Ingest is 90× cheaper than Ingest-heavy, and makes the query 403× faster than a query-optimized system (NoScope). On the other hand, Focus-Opt-Query reduces query latency even more (by 698×) but it is still 53× cheaper than Ingest-heavy. As these points in the design space are all good options compared to the baselines, such flexibility enables a user to tailor Focus for different contexts. For example, a camera that requires fast turnaround time for queries can use Focus-Opt-Query, while a video stream that will be queried rarely would choose Focus-Opt-Ingest to reduce the amount of wasted ingest cost in exchange for longer query latencies.

Figure 12 shows the (I, Q) values for both Focus-Opt-Ingest (Opt-I) and Focus-Opt-Query (Opt-Q) for the representative videos. As the figure shows, the flexibility to make different trade-offs exists in most other videos. On average, Focus-Opt-Ingest is 65× (up to 96×) cheaper than Ingest-heavy in ingest cost while providing 100× (up to 443×) faster queries. Focus-Opt-Query makes queries 202× (up to 698×) faster with a higher ingest cost (53× cheaper than Ingest-heavy). Note that there is no fundamental limitation on the spread between Focus-Opt-Query and Focus-Opt-Ingest as we can expand the search space for ingest-time CNNs to further optimize ingest cost at the expense of query latency (or vice versa). We conclude that Focus enables flexibly optimizing for ingest cost or query latency for application's needs.

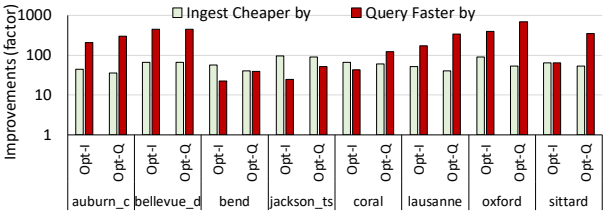


Figure 12: Ingest cost vs. query latency trade-off

It is worth noting that the fraction of videos that get queried can affect the applicability of Focus, especially in the case where only a tiny fraction of videos gets queried. While Focus-Opt-Ingest can save the ingest cost by up to 96×, it can be more costly than any purely query-time-only solution if the fraction of videos that gets queried is less than $\frac{1}{96} \approx 1\%$. In such a case, a user can still use Focus to significantly reduce query latency, but the cost of Focus can be higher than query-time-only solutions.

6.5. Sensitivity to Recall/Precision Target

Figure 13 illustrates Focus' reduction in query latency compared to the baselines under different recall/precision targets. Other than the default 99% recall and precision target, we evaluate both Focus and NoScope with two lower targets, 97% and 95%.

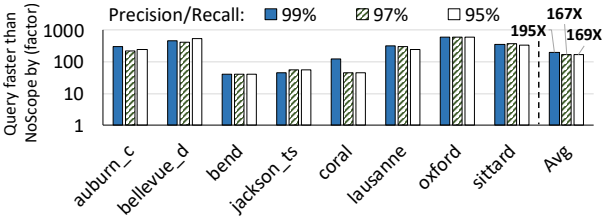


Figure 13: Sensitivity of query latency reduction to recall/precision target

We observe that with lower accuracy targets, the query latency improvement decreases slightly for most videos, while the ingest cost improvement does not change much (not graphed). The ingest cost is not sensitive to the accuracy target because Focus still runs similar ingest CNNs. NoScope can however apply more aggressive query-time optimization to reduce query latency given lower accuracy targets. This decreases Focus' improvement over NoScope for several videos. On average, Focus is faster than NoScope in query latency by 195×, 167×, and 169× with recall/precision of 99%, 97%, and 95%, respectively. We conclude that Focus' techniques can achieve significant improvements on query latency, irrespective of recall/precision targets.

6.6. Sensitivity to Object Class Numbers

We use the 1000 object classes in the ImageNet dataset [73] to study the sensitivity of Focus' performance to the number of object classes (compared to the 80 default object classes in the COCO [60] dataset). Our result shows that Focus is 15× faster (on average) in query latency and 57× cheaper (on average) in ingest cost than the baseline systems, while achieving 99% recall and precision. We observe that the query latency improvements with 1000 object classes is lower than the ones with 80 object classes. The reason is that ingest-time CNNs are less accurate on more object classes, and we need to select a larger K to achieve the target recall. Nonetheless, the improvements of Focus are robust with more object classes as Focus is over one order of magnitude faster than the baseline systems when differentiating 1000 object classes.

7. Other Applications

Applications that leverage CNNs to process large and continuously growing data share similar challenges as Focus. Examples of such applications are:

1) Video and audio. Other than querying for objects, many emerging video applications are also based on CNNs, such as event detection (e.g., [90]), emotion recognition (e.g., [49]), video classification (e.g., [52]), and face recognition (e.g., [74]). Audio applications such as speech recognition (e.g., [19]) are also based on CNNs.

2) Bioinformatics and geoinformatics. Many bioinformatics and geoinformatics systems leverage CNNs to process a large dataset, such as anomaly classification in biomedical imaging (e.g., [57, 72]), information decoding in biomedical signal recordings (e.g., [82]), and pattern recognition in satellite imagery (e.g., [20, 35]).

Naturally, these applications need to answer user-specified queries, such as "find all brain signal recordings with a particular perception" or "find all audio recordings with a particular keyword". Supporting these queries faces similar challenges to Focus, as a system either: (i) generates a precise index at ingest time, which incurs *high cost*; or (ii) does most of the heavy-lifting at query time, which results in *high query latency*. Hence, Focus' architecture offers a low-cost and low-latency option: building an approximate index with cheap CNNs at ingest time and generating precise results based on the approximate index at query time. While the indexing structure may need to be adapted to different applications, we believe Focus' architecture and techniques can benefit many of these emerging applications.

8. Related Work

To our knowledge, Focus is the first system that offers low-cost and low-latency queries for CNN-based object detection in videos by effectively splitting query-processing work between ingest time and query time. We discuss key works related to our system.

1) Cascaded classification. Various works in vision research propose speeding up classification by cascading a

series of classifiers. Viola et al. [88] is the earliest work that cascades a series of classifiers (from the simplest to the most complicated) to quickly disregard regions in an image. Many improvements follow (e.g., [58, 91, 92]). CNNs are also cascaded (e.g., [29, 43, 56, 83]) to reduce object detection latency. Our work is different in two major ways. First, we *decouple* the compressed CNN from the GT-CNN, which enables us to choose from a wider range of ingest-time CNNs and thus enables better trade-offs between ingest cost and query latency, a key aspect of our work. Second, we cluster similar objects using CNN features to eliminate redundant work, which is an effective technique for video streams.

2) Context-specific model specialization. Context-specific specialization of models can improve accuracy [63] or speed up inference [43, 51, 75]. Among these, the closest to our work is NoScope [51]. NoScope optimizes for the specified class at query-time using lightweight binary classifiers. In contrast, Focus' architecture splits work between ingest and query times, leading to two orders of magnitude lower latency (§6). To achieve these gains, Focus uses techniques to index *all possible classes* at ingest-time, and thus can handle any class that will get queried in the future. Focus' indexing is especially effective for less frequent object classes, which is arguably of more interest for video querying systems.

3) Stream processing systems. Systems for general stream data processing (e.g., [1, 18, 22, 25, 31, 32, 61, 66, 86, 87, 95]) and specific to video stream analytics (e.g., [96]) mainly focus on general stream processing challenges such as load shedding, fault tolerance, distributed execution, or limited network bandwidth. In contrast, our work is specific to querying on recorded video data with ingest and query trade-offs, and, thus, mostly orthogonal. Focus coud be integrated with one of these general stream processing systems.

4) Video indexing and retrieval. A large body of work in multimedia and information retrieval research proposes various content-based video indexing and retrieval techniques to facilitate queries on videos (e.g., [46, 55, 80, 81]). Among them, most works focus on indexing videos for different types of queries, such as shot boundary detection (e.g., [94]), semantic video search (e.g., [33, 37, 41]), video classification (e.g., [27]), spatio-temporal information-based video retrieval (e.g., [38, 70]) or subsequence similarity search (e.g., [76, 97]). Some works (e.g., [36, 79]) focus on the query interface to enable querying by keywords, concepts, or examples. These works are largely orthogonal to our work because we focus on reducing *cost and latency* of CNN-based video queries, not on creating an indexing structure for new query types or query interfaces. We believe our approach of splitting ingest-time and query-time work can be extended to many different types of video queries (§7).

5) Database indexing. Using index structures to reduce query latency [77] is a commonly-used technique in conventional databases (e.g., [26, 54]), key-value databases (e.g., [62]), similarity search (e.g., [39, 40]), graph queries (e.g., [93]), genome analysis (e.g., [21, 89]), and many others. Our Ingest-heavy and Ingest-NoScope baselines are also examples that index all video frames at ingest time. While queries are naturally faster with these baselines, they are too costly and are potentially wasteful for large-scale videos. In contrast, our work offers new trade-off options between ingest cost and query latency by creating low-cost approximate indexes at ingest time and retaining high accuracy with little work at query time.

9. Conclusion

Answering queries of the form, *find me frames that contain objects of class X*, is an important workload on recorded video datasets. Such queries are used by analysts and investigators for various immediate purposes, and it is crucial to answer them with low latency and low cost. We present Focus, a system that flexibly divides the query processing work between ingest time and query time. Focus performs low-cost ingest-time analytics on live video that later facilitates low-latency queries on the recorded videos. At ingest time, Focus uses cheap CNNs to construct an *approximate index* of all possible object classes in each frame to retain high recall. At query time, Focus leverages this approximate index to provide low latency, but compensates for the lower precision by judiciously using expensive CNNs. This architecture enables orders-of-magnitude faster queries with only a small investment at ingest time, and allows flexibly trading off ingest cost and query latency. Our evaluations using real-world videos from traffic, surveillance, and news domains show that Focus reduces ingest cost on average by 48× (up to 92×) and makes queries on average 125× (up to 607×) faster compared to state-of-the-art baselines. We conclude that Focus' architecture and techniques make it a highly practical and effective approach to querying large video datasets. We hope that the ideas and insights behind Focus can be applied to designing efficient systems for many other forms of querying on large and continuously-growing datasets in many domains, such as audio, bioinformatics, and geoinformatics.

Acknowledgments

We thank our shepherd, Andrew Warfield, and the anonymous OSDI reviewers for their valuable and constructive suggestions. We acknowledge the support of our industrial partners: Google, Huawei, Intel, Microsoft, and VMware. This work is supported in part by NSF and Intel STC on Visual Cloud Systems (ISTC-VCS).

References

[1] Apache Storm. http://storm.apache.org/index.html.

[2] Avigilon. http://avigilon.com/products/.

[3] City Cam, WebcamSittard: Town Square Sittard (NL). https://www.youtube.com/watch?v=iKxhsl3rurA.

[4] City of Auburn North Ross St and East Magnolia Ave. https://www.youtube.com/watch?v=cjuskMMYlLA.

[5] City of Auburn Toomer's Corner Webcam. https://www.youtube.com/watch?v=yJAk_FozAmI.

[6] Genetec. https://www.genetec.com/.

[7] Greenwood Avenue Bend, Oregon. https://www.youtube.com/watch?v=YqyERQwXA3U.

[8] Jackson Hole Wyoming USA Town Square. https://www.youtube.com/watch?v=K-F4CeVsWHA.

[9] L$\hat{2}$ Norm. http://mathworld.wolfram.com/L2-Norm.html.

[10] Lausanne, Place de la Palud. https://www.youtube.com/watch?v=7uF7DsUQ9vc.

[11] MongoDB. https://www.mongodb.com/.

[12] Nvidia Tesla P100. http://www.nvidia.com/object/tesla-p100.html.

[13] Opencv 3.4. http://opencv.org/opencv-3-4.html.

[14] OpenCV GPU-accelerated computer vision. https://docs.opencv.org/2.4/modules/gpu/doc/gpu.html.

[15] Oxford Martin School Webcam - Broad Street, Oxford. https://www.youtube.com/watch?v=Qhq4vQdfrFw.

[16] Top Video Surveillance Trends for 2016. https://technology.ihs.com/api/binary/572252.

[17] Wikipedia: Pareto efficiency. https://en.wikipedia.org/wiki/Pareto_efficiency.

[18] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[19] O. Abdel-Hamid, A. Mohamed, H. Jiang, and G. Penn. Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition. In *ICASSP*, 2012.

[20] A. Albert, J. Kaur, and M. C. Gonzalez. Using convolutional networks and satellite imagery to identify patterns in urban environments at a large scale. In *SIGKDD*, 2017.

[21] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 2009.

[22] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *DM-SSP*, 2006.

[23] A. Babenko and V. S. Lempitsky. Aggregating deep convolutional features for image retrieval. In *ICCV*, 2015.

[24] A. Babenko, A. Slesarev, A. Chigorin, and V. S. Lempitsky. Neural codes for image retrieval. In *ECCV*, 2014.

[25] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. MacroBase: Prioritizing attention in fast data. In *SIGMOD*, 2017.

[26] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET*, 1970.

[27] D. Brezeale and D. J. Cook. Automatic video classification: A survey of the literature. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 2008.

[28] S. Brutzer, B. Höferlin, and G. Heidemann. Evaluation of background subtraction techniques for video surveillance. In *CVPR*, 2011.

[29] Z. Cai, M. J. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *ICCV*, 2015.

[30] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SIAM International Conference on Data Mining*, 2006.

[31] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - A new class of data management applications. In *VLDB*, 2002.

[32] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

[33] S. Chang, W. Ma, and A. W. M. Smeulders. Recent advances and challenges of semantic image/video search. In *ICASSP*, 2007.

[34] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788, 2015.

[35] G. Cheng, Y. Wang, S. Xu, H. Wang, S. Xiang, and C. Pan. Automatic road detection and centerline extraction via cascaded end-to-end convolutional neural network. *IEEE Trans. Geoscience and Remote Sensing*, 2017.

[36] M. G. Christel and R. M. Conescu. Mining novice user activity with TRECVID interactive retrieval tasks. In *CIVR*, 2006.

[37] S. Dagtas, W. Al-Khatib, A. Ghafoor, and R. L. Kashyap. Models for motion-based video indexing and retrieval. *IEEE Trans. Image Processing*, 2000.

[38] M. E. Dönderler, Ö. Ulusoy, and U. Güdükbay. Rule-based spatiotemporal query processing for video databases. *VLDB J.*, 2004.

[39] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[40] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[41] A. Hampapur. Semantic video indexing: Approach and issue. *SIGMOD Record*, 1999.

[42] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

[43] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.

[44] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[45] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[46] W. Hu, N. Xie, L. Li, X. Zeng, and S. J. Maybank. A survey on visual content-based video indexing and retrieval. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 2011.

[47] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[48] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[49] S. E. Kahou, C. J. Pal, X. Bouthillier, P. Froumenty, Ç. Gülçehre, R. Memisevic, P. Vincent, A. C. Courville, Y. Bengio, R. C. Ferrari, M. Mirza, S. Jean, P. L. Carrier, Y. Dauphin, N. Boulanger-Lewandowski, A. Aggarwal, J. Zumer, P. Lamblin, J. Raymond, G. Desjardins, R. Pascanu, D. Warde-Farley, A. Torabi, A. Sharma, E. Bengio, K. R. Konda, and Z. Wu. Combining modality specific deep neural networks for emotion recognition in video. In *ICMI*, 2013.

[50] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope project website. `https://github.com/stanford-futuredata/noscope`.

[51] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing deep CNN-based queries over video streams at scale. *PVLDB*, 2017.

[52] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[54] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 1981.

[55] M. S. Lew, N. Sebe, C. Djeraba, and R. Jain. Content-based multimedia information retrieval: State of the art and challenges. *TOMCCAP*, 2006.

[56] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *CVPR*, 2015.

[57] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen. Medical image classification with convolutional neural network. In *ICARCV*, 2014.

[58] R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. In *ICIP*, 2002.

[59] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017.

[60] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. In *ECCV*, 2014.

[61] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[62] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[63] A. Mhalla, H. Maâmatou, T. Chateau, S. Gazzah, and N. E. B. Amara. Faster R-CNN scene specialization with a sequential monte-carlo framework. In *DICTA*.

[64] Microsoft. The microsoft Cognitive Toolkit. `https://www.microsoft.com/en-us/cognitive-toolkit/`.

[65] L. O'Callaghan, N. Mishra, A. Meyerson, and S. Guha. Streaming-data algorithms for high-quality clustering. In *ICDE*, 2002.

[66] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *NSDI*, 2014.

[67] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: An astounding baseline for recognition. In *CVPR Workshops*, 2014.

[68] J. Redmon and A. Farhadi. YOLO9000: Better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[69] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

[70] W. Ren, S. Singh, M. Singh, and Y. S. Zhu. State-of-the-art on spatio-temporal information-based video retrieval. *Pattern Recognition*, 2009.

[71] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. FitNets: Hints for thin deep nets. *CoRR*, abs/1412.6550, 2014.

[72] H. R. Roth, L. Lu, J. Liu, J. Yao, A. Seff, K. M. Cherry, L. Kim, and R. M. Summers. Improving computer-aided detection using convolutional neural networks and random view aggregation. *IEEE Trans. Med. Imaging*, 2016.

[73] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *IJCV*, 2015.

[74] F. Schroff, D. Kalenichenko, and J. Philbin. FaceNet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.

[75] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *CVPR*, 2017.

[76] H. T. Shen, B. C. Ooi, and X. Zhou. Towards effective indexing for very large video sequence database. In *SIGMOD*, 2005.

[77] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[78] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[79] C. Snoek, K. E. A. van de Sande, O. de Rooij, B. Huurnink, E. Gavves, D. Odijk, M. de Rijke, T. Gevers, M. Worring, D. Koelma, and A. W. M. Smeulders. The mediamill TRECVID 2010 semantic video search engine. In *TRECVID 2010 workshop participants notebook papers*, 2010.

[80] C. Snoek and M. Worring. Multimodal video indexing: A review of the state-of-the-art. *Multimedia Tools Appl.*, 2005.

[81] C. G. M. Snoek and M. Worring. Concept-based video retrieval. *Foundations and Trends in Information Retrieval*, 2009.

[82] S. Stober, D. J. Cameron, and J. A. Grahn. Using convolutional neural networks to recognize rhythm stimuli from electroencephalography recordings. In *NIPS*, 2014.

[83] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *CVPR*, 2013.

[84] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[85] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[86] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *VLDB*, 2007.

[87] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, 2006.

[88] P. A. Viola and M. J. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.

[89] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan. Accelerating read mapping with FastHASH. *BMC Genomics*, 2013.

[90] Z. Xu, Y. Yang, and A. G. Hauptmann. A discriminative CNN video representation for event detection. In *CVPR*, 2015.

[91] Z. E. Xu, M. J. Kusner, K. Q. Weinberger, and M. Chen. Cost-sensitive tree of classifiers. In *ICML*, 2013.

[92] Q. Yang, C. X. Ling, X. Chai, and R. Pan. Test-cost sensitive classification on data with missing values. *IEEE Trans. Knowl. Data Eng.*, 2006.

[93] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDB J.*, 2013.

[94] J. Yuan, H. Wang, L. Xiao, W. Zheng, J. Li, F. Lin, and B. Zhang. A formal study of shot boundary detection. *IEEE Trans. Circuits Syst. Video Techn.*, 2007.

[95] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[96] H. Zhang, G. Ananthanarayanan, P. Bodík, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, 2017.

[97] X. Zhou, X. Zhou, L. Chen, and A. Bouguettaya. Efficient subsequence matching over large video databases. *VLDB J.*, 2012.

Nickel: A Framework for Design and Verification of Information Flow Control Systems

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney,
James Bornholt, Emina Torlak, Xi Wang
University of Washington

Abstract

Nickel is a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface, which can be exploited to circumvent the enforcement of information flow policies. Nickel provides a formulation of noninterference amenable to automated verification, allowing developers to specify an intended policy of permitted information flows. It invokes the Z3 SMT solver to verify that both an interface specification and an implementation satisfy noninterference with respect to the policy; if verification fails, it generates counterexamples to illustrate covert channels that cause the violation.

Using Nickel, we have designed, implemented, and verified NiStar, the first OS kernel for decentralized information flow control that provides (1) a precise specification for its interface, (2) a formal proof that the interface specification is free of covert channels, and (3) a formal proof that the implementation preserves noninterference. We have also applied Nickel to verify isolation in a small OS kernel, NiKOS, and reproduce known covert channels in the ARINC 653 avionics standard. Our experience shows that Nickel is effective in identifying and ruling out covert channels, and that it can verify noninterference for systems with a low proof burden.

1 Introduction

Operating systems often provide information flow control mechanisms to improve application security. These mechanisms enforce policies ranging from strict isolation to more flexible models using labels [12, 60]. By tracking and mediating data access, they aim to regulate the propagation of information among applications to provide secrecy and integrity guarantees.

Malicious applications can circumvent information flow control systems by encoding and transferring information indirectly, such as through temporary files, process names, or CPU and memory usage [47]. Many such *covert channels* exist not only in the POSIX interface but also in specialized information flow control systems (see §2 for a survey). For example, Krohn et al. [45] have described covert channels in Asbestos [15] that allow applications to leak data at a high bandwidth. Covert channels in the interface are critical flaws as *no* secure implementation of such an interface can exist [44]. Eliminating these channels at the interface level is thus a key challenge in the design of information flow control systems.

Even if an interface specification is free of covert channels, it remains challenging to correctly implement the system—incorrect or missing checks will invalidate the guarantees of information flow control. For instance, both KLEE [6: §5.3] and STACK [78: §6.1] have found such bugs in HiStar [82]. As another example, the implementation of Flume [45] relies on the Linux kernel, which is likely to contain bugs given its complexity [8, 51, 63].

This paper presents Nickel, a framework for systematically eliminating covert channels from such systems through formal verification of *noninterference*. Noninterference is a general security criterion that has been extensively studied in prior work [24, 53, 67]. Intuitively, given two mutually distrustful threads between which information flow is prohibited, noninterference requires the output of operations in one thread to be independent of operations in the other thread. This restriction ensures that a malicious thread can neither infer secrets nor influence the execution path of another thread via operations defined in the interface; any violation indicates a covert channel. However, applying noninterference to reason about an interface requires considering the precise behavior of each operation as well as the interaction of all pairs of operations [38, 39], which is non-trivial. Nickel helps automate this reasoning using an SMT solver such as Z3 [11].

Nickel introduces both a formulation of noninterference and new proof strategies that are amenable to automated verification. It asks developers to specify a concise and intuitive policy that describes permitted flows in a system, and checks whether a given interface specification satisfies noninterference for that policy. Furthermore, it extends our previous work on push-button verification [62, 70] to check whether a given implementation preserves noninterference through refinement. Verifying both an interface and an implementation this way incurs a low proof burden (see §8). An additional advantage of automated reasoning is that Nickel will provide a *counterexample* when it finds a covert channel in either the interface or the implementation, which is valuable for debugging and revising the design.

We have applied Nickel to three systems. The foremost is NiStar, a new OS kernel with provably secure decentralized information flow control (DIFC) [60]. DIFC is a flexible mechanism that allows applications to express powerful policies, but this flexibility makes it challenging to analyze covert channels and security implications of DIFC systems [44]. Inspired by HiStar [82], NiStar provides DIFC support through a small number of kernel object types. Unlike HiStar, however, NiStar provides a formal proof that both its interface and implementation satisfy noninterference, ruling out covert channels in the design. To the best of our knowledge, NiStar is the first formally verified DIFC OS kernel.

To demonstrate Nickel's applicability to a broader set of systems, we have used Nickel to verify NiKOS, an OS kernel that mirrors mCertiKOS [10] to enforce process isolation. We have also applied Nickel to formalize and analyze the specification of the communication interface of ARINC 653 [1], an industrial avionics standard. Nickel was able to reproduce the three covert channels in ARINC 653 previously reported by Zhao et al. [86].

Nickel reasons about sequential (uniprocessor) systems and provides no guarantees in concurrent settings. It focuses on eliminating covert channels inherent in the interface; physical effects (e.g., timing, sound, and energy) that are not captured by the interface specification are beyond the scope of this paper. We discuss these limitations further in §3.5.

In summary, this paper makes three contributions: (1) a formulation of noninterference and proof strategies amenable to automated reasoning; (2) the Nickel framework for verifying noninterference for the interface and implementation of information flow control systems; and (3) the formal specifications of three systems, including the first formally verified DIFC OS kernel.

The rest of this paper is organized as follows. §2 surveys common patterns of covert channels in interfaces. §3 formalizes noninterference and introduces theorems for proving noninterference. §4 gives an overview of the development workflow using Nickel. §5 presents guidelines for interface design. §6 describes the design, implementation, and verification of DIFC in NiStar. §7 describes the verification of isolation in NiKOS and ARINC 653. §8 reports our experience with using Nickel. §9 relates Nickel to prior work. §10 concludes.

2 Covert channels in interfaces

Nickel's main goal is to help developers identify and eliminate covert channels in the interface of an information flow control system. This section surveys common examples of covert channels and shows how to apply noninterference to understand them.

Consider two threads T_1 and T_2 that are prohibited from communicating as per the information flow policy. What kinds of interface operations can be exploited by the two threads to collude and bypass the policy (or equivalently, allow an adversarial T_2 to infer secret information from an uncooperative T_1)? As a simple example, if an operation introduces shared memory locations that both threads can read and write, then the two threads can use these memory locations as covert channels to transfer information. Unintended covert channels, however, are often subtle and difficult to spot, as detailed next.

Resource names. Resource names, such as thread identifiers, page numbers, and port numbers, can be used to encode information. Consider a system call `spawn` that creates new threads with sequential identifiers. Thread T_2 first spawns a thread with an identifier, say, 3; the other thread T_1 then spawns x times, creating threads with identifiers from 4 to $x+3$; and thread T_2 spawns another thread, whose identifier will be $x+3+1$. In doing so, thread T_2 learns the secret x from T_1 through the difference of the identifiers of the two threads it has created [10: §5].

Resource exhaustion. Suppose that the system has a total of N pages shared by all threads. Thread T_1 first allocates $N-1$ pages, and encodes a one-bit secret by either allocating the last page or not. The other thread T_2 then tries to allocate one page and learns the secret based on whether the allocation succeeds [82: §3]. This covert channel is effective especially when a resource is limited and can be easily exhausted.

Statistical information. A thread's world-readable information, such as its name, number of open file descriptors, and CPU and memory usage, can be used to encode secret data or by adversarial threads to learn secrets [35, 85]. For example, if thread T_1's memory usage is accessible to another thread T_2 through `procfs` or system calls, T_1 could leak a secret x by allocating x pages.

Error handling. Error handling is known to be prone to information leakage [54], such as the TENEX password-guessing attack using page faults [48] and the POODLE attack against TLS [55]. As an example, consider a system call for querying the status of a page, which returns `-ENOENT` if the given page is free and `-EACCES` if the page is in use but not accessible by the calling thread. Thread T_1 encodes a one-bit secret by allocating a particular page or not; the other thread T_2 queries the status of that page and learns the secret based on whether the error code is `-ENOENT` or `-EACCES`.

Scheduling. Suppose an OS kernel uses a round-robin scheduler that distributes time slices evenly among threads. Thread T_1 encodes a secret by forking a number of threads (e.g., a fork bomb), which causes the other thread T_2 to observe the reduction of time allocated for itself and learn the secret from T_1; alternatively, T_2 can

continuously ping a remote server, which will learn the secret from the time between pings [82: §9]. Access to only logical time suffices for such covert channels.

External devices and services. Suppose the system allows threads to communicate with external devices and services. Thread T_1 can write secret data to the registers of a device, or encode the secret as the frequency of accessing a device or even through a service bill [47]; the other thread T_2 can then retrieve the secret at a later time from the same device or service.

Mutable labels. Many information flow control systems express security policies by assigning *labels* to objects. Label changes complicate such systems and can lead to covert channels [12]. As an example, consider a system where each thread can be labeled as either tainted or untainted. The system enforces a tainting policy: a tainted thread cannot transfer information to an untainted thread without tainting it. To enforce this policy, the system raises the label of an untainted thread to tainted when another tainted thread sends data to it. Suppose thread T_1 is tainted and thread T_2 is untainted. To bypass the policy, T_2 first spawns an untainted helper thread H. T_1 encodes a one-bit secret by choosing whether to send data to taint H, which in turn chooses to send data to T_2 *only* if it is untainted and do nothing otherwise. In this way, T_2 learns the secret from T_1 by whether it receives data from H, without becoming tainted itself [44: §3].

2.1 Applying noninterference

Given two threads T_1 and T_2 that are prohibited from communicating with each other, noninterference states that the output of operations in one thread should not be affected by whether operations in the other thread occur. Now we will show how to apply noninterference to uncover covert channels.

Take the `spawn` system call as an example, which returns sequential thread identifiers and introduces a covert channel due to resource names. Figure 1 illustrates this channel. We denote an action of invoking a system call as a left half-circle `spawn` and its return value as a right half-circle `3`. We use different colors to distinguish system calls from different threads: $\texttt{spawn}_1$ in T_1; $\texttt{spawn}_0$ and $\texttt{spawn}_2$ in T_2.

We apply noninterference to uncover the covert channel introduced by `spawn` in three steps. First, construct a trace of actions from both threads, for instance, $\texttt{spawn}_0$ $\texttt{spawn}_1$ $\texttt{spawn}_2$. Assume that the corresponding return values (i.e., outputs) are `3` `4` `5`, as `spawn` sequentially allocates identifiers. Second, to examine possible effects of T_1 on T_2, construct a new trace that *purges* the actions from T_1 and retains the actions only from T_2, resulting in $\texttt{spawn}_0$ $\texttt{spawn}_2$. Third, replay this purged trace to the system, obtaining a new sequence of outputs `3` `4`. This

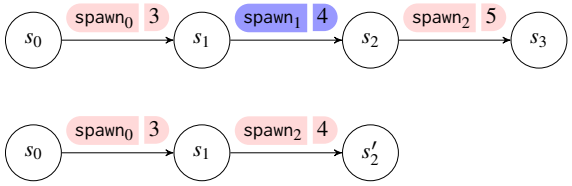


Figure 1: The output of $\texttt{spawn}_2$ changes from 5 in the original trace (first row) to 4 in the purged trace (second row), indicating a covert channel. Circles denote states, arrows denote state transitions, left half-circles denote actions, and right half-circles denote outputs.

sequence differs from the original output of the same actions, which is `3` `5`. The change of output in T_2 (in particular, the return value of $\texttt{spawn}_2$) caused by an action in T_1 violates noninterference, indicating a covert channel with which T_1 may transfer information to T_2. On the other hand, with a version of `spawn` that does not introduce a covert channel, the outputs of T_2's actions in the purged and original traces would be the same.

One can similarly apply noninterference to uncover the other covert channels described in this section. The challenge is to find a trace of actions that manifests the covert channel, and if there are no such channels, to exhaustively show that no trace violates noninterference. Nickel automates this task using formal verification techniques, as we will describe next.

3 Proving noninterference

This section formalizes the notion of noninterference used in Nickel and presents the main theorems that enable Nickel to prove noninterference for systems.

First, we address how to specify the intended policy of an information flow control system. The policy is trusted as the top-level specification of the system, which will be used to catch and fix potential covert channels in both the interface specification and the implementation (§3.1).

Next, we give a formal definition of noninterference in terms of traces of actions, which precisely captures whether an interface specification satisfies a given policy (§3.2).

To prove noninterference for an interface specification, Nickel introduces an unwinding verification strategy that requires reasoning only about individual actions, rather than traces of actions (§3.3). To extend the guarantee of noninterference to an implementation, Nickel introduces a restricted form of refinement that preserves noninterference (§3.4). Both strategies are amenable to automated verification using an SMT solver.

We end this section with a discussion of the limitations of the Nickel approach (§3.5).

3.1 Policy

We model the execution of a system as a state machine in a standard way [67]. A system $\mathcal{M}$ is defined as a tuple

$\langle A, O, S, \texttt{init}, \texttt{step}, \texttt{output}\rangle$, where A is the set of actions, O is the set of output values, S is the set of states, $\texttt{init}$ is the initial state, $\texttt{step} : S \times A \to S$ is the state-transition function, and $\texttt{output} : S \times A \to O$ is the output function.

An action transitions the system from state to state. In the context of an OS, an action can be either a user-space operation (e.g., memory access), or the handling of a trap due to system calls, exceptions, or scheduling. Each action consists of an operation identifier (e.g., the system call number) and arguments. We write $\texttt{output}(s, a)$ and $\texttt{step}(s, a)$ to denote the output value (e.g., the return value of a system call) and the next state, respectively, for the state s and action a. Actions are considered to be atomic; for instance, we assume that an OS kernel executes each trap handler with interrupts disabled on a uniprocessor system [40, 62].

A *trace* is a sequence of actions. We use $\texttt{run}(s, tr)$ to denote the state produced by executing each action in trace tr starting from state s. The $\texttt{run}$ function is defined as follows:

$$\texttt{run}(s, \epsilon) \coloneqq s$$
$$\texttt{run}(s, a \circ tr) \coloneqq \texttt{run}(\texttt{step}(s, a), tr).$$

Here, ϵ denotes the empty trace, and $a \circ tr$ denotes the concatenation of action a and trace tr.

Definition 1 (Information Flow Policy)**.** A policy $\mathcal{P}$ for system $\mathcal{M}$ is defined as a tuple $\langle D, \rightsquigarrow, \texttt{dom}\rangle$, where D is the set of *domains*, $\rightsquigarrow \subseteq (D \times D)$ is the can-flow-to relation between two domains, and the function $\texttt{dom} : A \times S \to D$ maps an action with a state to a domain.

Intuitively, a domain is an abstract representation of the exercised authority of an action. A policy associates each action a performed from state s with a domain, denoted by $\texttt{dom}(a, s)$; the can-flow-to relation $\rightsquigarrow$ defines permitted information flows among these domains. The goal of a policy is to explicitly specify permitted flows and ensure that any trace of actions, given their specifications, will *not* lead to covert channels that enable unintended flows and violate the policy.

Below we show the policies for two example systems. We write $u \rightsquigarrow v$ and $u \not\rightsquigarrow v$ to mean $(u, v) \in \rightsquigarrow$ and $(u, v) \notin \rightsquigarrow$, respectively.

Example (Tainting)**.** Consider the label-based system mentioned in §2: it has a number of threads, where the label of each thread is either tainted or untainted. The system enforces a tainting policy as depicted in Figure 2. The policy permits information flow from untainted threads to either untainted or tainted threads, and between two tainted threads, but it prohibits untainted threads from directly communicating with tainted ones.

For this policy, we designate $\{tainted, untainted\}$ as the set of domains. The can-flow-to relation consists of the following three permitted flows: $tainted \rightsquigarrow tainted$, $untainted \rightsquigarrow untainted$, and $untainted \rightsquigarrow tainted$. The $\texttt{dom}$ function returns the label of the thread currently running. NiStar employs a more sophisticated version of this policy using DIFC (see §6).

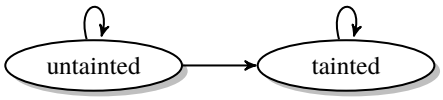


Figure 2: The tainting policy: information cannot flow from tainted threads to untainted threads.

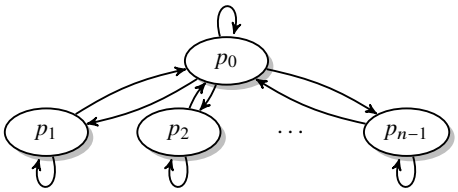


Figure 3: The isolation policy of NiKOS: information cannot flow between any two of the regular processes $p_1, p_2, \ldots, p_{n-1}$ (except through the scheduler p_0 indirectly).

Example (Isolation)**.** Consider a Unix-like kernel with n processes: a special scheduler process p_0, and regular processes $p_1, p_2, \ldots, p_{n-1}$. The system enforces a process isolation policy as depicted in Figure 3, which permits information flows from a process to itself, from the scheduler to any process, and from any process to the scheduler; no information flow is permitted between any two regular processes except indirectly through the scheduler [10].

To specify this isolation policy, we designate the processes $\{p_0, p_1, \ldots, p_{n-1}\}$ as the set of domains, where p_0 is the scheduler. The can-flow-to relation consists of the permitted flows $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$, for all $i \in [0, n-1]$. The $\texttt{dom}$ function returns the currently running process as the domain for system call actions, and returns the scheduler p_0 as the domain for context switching actions. NiKOS employs this policy (see §7).

We highlight two features in our policy definition (Definition 1). First, it allows the can-flow-to relation $\rightsquigarrow$ to be *intransitive* [67]. For instance, the isolation policy permits processes p_1 and p_2 to communicate through the scheduler, but prohibits them from communicating directly with each other. In other words, $p_1 \rightsquigarrow p_0$ and $p_0 \rightsquigarrow p_2$ do *not* have to imply $p_1 \rightsquigarrow p_2$, though that would also be accepted by Nickel if it were the intended policy.

This generality enables Nickel to support a broad range of policies, as practical systems often need *downgrading* operations (e.g., intentional declassification and endorsement) [49]. As a simple example, a system may prefer to have an untrusted application send data to an encryption program, which in turn is permitted to reach the network, while the application itself is prohibited from sending

$$\mathtt{sources}(\epsilon, u, s) \coloneqq \{u\}$$

$$\mathtt{sources}(a \circ tr, u, s) \coloneqq \mathtt{sources}(tr, u, \mathtt{step}(s, a)) \cup \begin{cases} \{\mathtt{dom}(a, s)\} & \text{if } \exists v \in \mathtt{sources}(tr, u, \mathtt{step}(s, a)).\ \mathtt{dom}(a, s) \rightsquigarrow v \\ \varnothing & \text{otherwise.} \end{cases}$$

Figure 4: $\mathtt{sources}(tr, u, s)$ is the set of domains that are allowed to influence domain u over a trace tr, starting from state s.

$$\mathtt{purge}(\epsilon, u, s) \coloneqq \{\epsilon\}$$

$$\mathtt{purge}(a \circ tr, u, s) \coloneqq \{a \circ tr' \mid tr' \in \mathtt{purge}(tr, u, \mathtt{step}(s, a))\} \cup \begin{cases} \varnothing & \text{if } \mathtt{dom}(a, s) \in \mathtt{sources}(a \circ tr, u, s) \\ \mathtt{purge}(tr, u, s) & \text{otherwise.} \end{cases}$$

Figure 5: $\mathtt{purge}(tr, u, s)$ is the set of all sub-traces of tr that retain the actions that are allowed to influence domain u, starting from state s.

data directly over the network. Such policies require intransitive can-flow-to relations [67, 80].

Second, in classical noninterference [24, 67], the dom function is state-independent ($A \to D$). The definition of dom used in Nickel is *state-dependent* ($A \times S \to D$). This extension is necessary for reasoning about many systems in which the domain (i.e., authority) of an action depends on the currently running thread or process [56, 68]. As we will show next, we have developed a definition of noninterference and theorems for proving noninterference that accommodate this extension.

3.2 Noninterference

Given a system and a policy for the system, what kind of action can violate the policy and introduce covert channels? As described in §2, to check for noninterference, one can construct a trace of actions, obtain a purged trace by removing actions from the original trace as per the policy, and compare the output of the corresponding actions in both traces—any change of output indicates a covert channel. Below we give a precise definition of noninterference that captures this intuition, in three steps.

First, suppose that a system has executed a trace tr to reach the state $\hat{s} = \mathtt{run}(\mathtt{init}, tr)$, and is about to perform action $\hat{a}$ next. To construct a purged trace of tr, we need to identify the actions that the policy permits to influence a domain u and therefore should be retained in the trace. This set is defined using the $\mathtt{sources}(tr, u, s)$ function shown in Figure 4, which returns the set of domains that can transfer information to domain u over trace tr from state s, either directly specified by the can-flow-to relation or indirectly through the domain of another intermediate action in the trace.

Second, to obtain a purged trace that retains the actions identified by sources, we define the $\mathtt{purge}(tr, u, s)$ function as shown in Figure 5. It returns the set of all sub-traces of tr where each action in the sources of u from state s has been retained; the actions whose domains are not identified by sources are optionally removed.

Third, let tr' denote a purged trace in the set $\mathtt{purge}(tr, \mathtt{dom}(\hat{a}, \hat{s}), \mathtt{init})$; like other traces in this set, tr' is obtained by retaining actions in trace tr that can transfer information to action $\hat{a}$. Now let's replay the purged trace tr' from the start, resulting in a new state $\hat{s}' = \mathtt{run}(\mathtt{init}, tr')$. If the system satisfies noninterference for the policy, then invoking $\hat{a}$ from state $\hat{s}$ should produce the same output as invoking $\hat{a}$ from state $\hat{s}'$.

Formally, we define noninterference as follows:

Definition 2 (Noninterference)**.** Given a system $\mathcal{M} = \langle A, O, S, \mathtt{init}, \mathtt{step}, \mathtt{output}\rangle$ and a policy $\mathcal{P} = \langle D, \rightsquigarrow, \mathtt{dom}\rangle$, $\mathcal{M}$ satisfies noninterference for $\mathcal{P}$ if and only if the following holds for any trace tr, action a, and purged trace $tr' \in \mathtt{purge}(tr, \mathtt{dom}(a, \mathtt{run}(\mathtt{init}, tr)), \mathtt{init})$:

$$\mathtt{output}(\mathtt{run}(\mathtt{init}, tr), a) = \mathtt{output}(\mathtt{run}(\mathtt{init}, tr'), a).$$

To ensure that our definition of noninterference is reasonable, we show two properties of this definition. First, recall that we use a state-dependent dom function; if dom is restricted to be state-independent, that is, $\mathtt{dom}(a, s) = \mathtt{dom}(a)$ holds for any a and s, then our definition reduces to classical noninterference [67], suggesting that our definition is a natural extension.

Second, a reasonable definition of noninterference should be *monotonic* [17]: a system satisfying noninterference for some policy should also satisfy noninterference for a more relaxed policy in which more flows are permitted. More formally, given two policies $\mathcal{P} = \langle D, \rightsquigarrow, \mathtt{dom}\rangle$ and $\mathcal{P}' = \langle D, \rightsquigarrow', \mathtt{dom}\rangle$, we say $\mathcal{P}'$ *contains* $\mathcal{P}$ to mean that any flow permitted by $\mathcal{P}$ is also permitted by $\mathcal{P}'$ (i.e., $\rightsquigarrow \subseteq \rightsquigarrow'$). We have proved the following monotonicity property as a sanity check on our definition of noninterference: if a system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$, then it also satisfies noninterference for any policy $\mathcal{P}'$ that contains $\mathcal{P}$.

3.3 Unwinding

It is difficult to directly apply Definition 2 to prove noninterference for a given system and policy, as it requires

$\mathcal{I}$ is a state invariant:

$\mathcal{I}(\texttt{init}) \wedge (\mathcal{I}(s) \Rightarrow \mathcal{I}(\texttt{step}(s,a)))$

$\overset{u}{\approx}$ is an equivalence relation:

$\overset{u}{\approx}$ is reflexive, symmetric, and transitive

$\overset{u}{\approx}$ is consistent with `dom`:

$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{dom}(a,s) = \texttt{dom}(a,t)$

$\overset{u}{\approx}$ is consistent with $\rightsquigarrow$:

$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \Rightarrow (\texttt{dom}(a,s) \rightsquigarrow u \Leftrightarrow \texttt{dom}(a,t) \rightsquigarrow u)$

output consistency:

$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{output}(s,a) = \texttt{output}(t,a)$

local respect:

$\mathcal{I}(s) \wedge \texttt{dom}(a,s) \not\rightsquigarrow u \Rightarrow s \overset{u}{\approx} \texttt{step}(s,a)$

weak step consistency:

$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{step}(s,a) \overset{u}{\approx} \texttt{step}(t,a)$

Figure 6: Unwinding conditions. Each formula is universally quantified over its free variables, such as domain u, action a, and states s and t.

reasoning about all possible traces. A standard approach is to define a set of *unwinding conditions*, which together imply noninterference but require reasoning only about individual actions. We generalize the classical unwinding conditions given by Rushby [67] to obtain an unwinding theorem that accommodates our state-dependent `dom` function and is amenable to automated verification. Proving noninterference using the unwinding theorem requires two extra inputs from developers: a *state invariant* and an *observational equivalence* relation, as described next.

A state invariant $\mathcal{I}$ [46] is a state predicate that must hold on all *reachable* states (i.e., the set of states produced by running any trace starting from the `init` state). The state invariant overapproximates the set of reachable states, as it may also hold for unreachable states. If the unwinding theorem holds for states satisfying $\mathcal{I}$, then it holds for all reachable states of the system. We use this overapproximation to enable automation: in contrast to reachability, which cannot be expressed in first-order logic, the state invariant can be both expressed and effectively checked with an SMT solver.

The next input required for the unwinding theorem is an observational equivalence relation $\approx \subseteq (D \times S \times S)$. The observational equivalence describes, for each domain, the set of states that appear to that domain to be indistinguishable. We write $\overset{u}{\approx}$ to mean the binary relation $\{(s,t) \mid (u,s,t) \in \approx\}$ relating all equivalent states for domain u, and $s \overset{u}{\approx} t$ to mean $(u,s,t) \in \approx$.

We then define the unwinding conditions of system $\mathcal{M}$ for policy $\mathcal{P}$, shown in Figure 6, and prove the following unwinding theorem:

Theorem 1 (Unwinding)**.** A system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$ if there exists a state invariant $\mathcal{I}$ and an observational equivalence relation $\approx$ for which the unwinding conditions in Figure 6 hold.

The unwinding theorem obviates the need to reason about traces to prove noninterference; instead, it suffices to show that the unwinding conditions hold for each action. This theorem enables Nickel to automate the checking using the Z3 SMT solver (see §4). Both the state invariant $\mathcal{I}$ and the observational equivalence relation $\approx$ are *untrusted*: any instances that satisfy the conditions are sufficient to establish noninterference.

We give some intuition behind the unwinding theorem. The first four conditions are natural: they ask for a reasonable state variant $\mathcal{I}$ and observational equivalence relation $\approx$ (i.e., $\overset{u}{\approx}$ should be an equivalence relation and be consistent with the policy). The remaining three conditions, *output consistency*, *local respect*, and *weak step consistency*, provide more hints to interface design, as follows. As a shorthand, we say "objects" to mean individual storage locations in the system state.

First, the output of an action should depend only on objects that the domain of the action can read. Restricting the output prevents an adversarial application from inferring information about system state via return values, such as the error-handling channel described in §2.

Second, if an action attempts to modify an object, the domain of the action should be able to write to that object, and its new value should depend only on the old value and objects that the domain of the action can read. This requirement prevents unintended flows while updating the system state, such as the resource-name channel introduced by `spawn` sequentially allocating identifiers.

Third, if an action attempts to create a new object, that new object should have equal or less authority than the domain of the action; similarly, if an object becomes newly readable after an action, then the domain of the action should have been able to read that object before the call. These restrictions preclude "runaway" authority—no action can arbitrarily increase the authority of its domain, or create an object more powerful than itself.

3.4 Refinement

Refinement is widely used for verifying systems: developers describe the intended system behavior as a high level, abstract specification and check that any behavior exhibited by a low level, concrete implementation is allowed by the specification. Refinement allows developers to reason about many properties of the system at the specification level, which is often simpler than reasoning about the implementation directly.

In our case, it would be ideal to prove noninterference (using the unwinding theorem) for an interface specification, and extend that guarantee to an implementation that refines the specification. However, it is well known that noninterference is generally *not* preserved under refinement [25, 52]; for example, the implementation may introduce extra stuttering steps that leak information. Nickel

supports a restricted form of refinement over state machines and policies. We show here that this refinement preserves noninterference as defined in §3.2.

Let's consider the following systems:

- $\mathcal{M}_1 = \langle A, O, S_1, \texttt{init}_1, \texttt{step}_1, \texttt{output}_1 \rangle$, and
- $\mathcal{M}_2 = \langle A, O, S_2, \texttt{init}_2, \texttt{step}_2, \texttt{output}_2 \rangle$.

These two systems share the set of actions A and the set of outputs O, but differ in the state spaces, as well as the state-transition and output functions. One may consider $\mathcal{M}_1$ as the specification and $\mathcal{M}_2$ as the implementation. We say that $\mathcal{M}_2$ is a *data refinement* of $\mathcal{M}_1$ to mean that they produce the same output for any trace [33, 46]. Data refinement is particularly useful for verifying systems with a well-defined interface, such as OS kernels [41, 62].

A standard way to prove data refinement of $\mathcal{M}_1$ by $\mathcal{M}_2$ is to ask developers to identify a data refinement relation $\propto \subseteq (S_2 \times S_1)$; we write $s_2 \propto s_1$ to mean $(s_2, s_1) \in \propto$. Let $\mathcal{I}_2$ denote a state invariant for $\mathcal{M}_2$. To prove that $\mathcal{M}_2$ is a data refinement of $\mathcal{M}_1$, it suffices to show that the following *refinement conditions* hold:

- $\texttt{init}_2 \propto \texttt{init}_1$.
- $\mathcal{I}_2(s_2) \wedge s_2 \propto s_1 \Rightarrow \texttt{step}_2(s_2, a) \propto \texttt{step}_1(s_1, a)$.
- $\mathcal{I}_2(s_2) \wedge s_2 \propto s_1 \Rightarrow \texttt{output}_2(s_2, a) = \texttt{output}_1(s_1, a)$.

Each formula is universally quantified over s_1, s_2, and a.

Given policies $\mathcal{P}_1 = \langle D, \rightsquigarrow, \texttt{dom}_1 \rangle$ and $\mathcal{P}_2 = \langle D, \rightsquigarrow, \texttt{dom}_2 \rangle$ for systems $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, we say that $\mathcal{P}_2$ is a *policy refinement* of $\mathcal{P}_1$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if and only if the following holds for any action a and trace tr: $\texttt{dom}_1(a, \texttt{run}_1(\texttt{init}_1, tr)) = \texttt{dom}_2(a, \texttt{run}_2(\texttt{init}_2, tr))$. Here $\texttt{run}_1$ and $\texttt{run}_2$ apply a trace starting from a given state for $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively (§3.2).

With these notions of data refinement and policy refinement, we have proved the following refinement theorem for noninterference:

Theorem 2 (Refinement)**.** Given two systems $\mathcal{M}_1$ and $\mathcal{M}_2$ and policy $\mathcal{P}$ for $\mathcal{M}_1$, $\mathcal{M}_2$ satisfies noninterference for any policy refinement of $\mathcal{P}$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if:

- there exists a state invariant $\mathcal{I}_1$ of system $\mathcal{M}_1$ and an observational equivalence relation $\approx$ for which the unwinding conditions of $\mathcal{M}_1$ for $\mathcal{P}$ hold; and
- there exists a state invariant $\mathcal{I}_2$ of system $\mathcal{M}_2$ and a data refinement relation $\propto$ for which the refinement conditions of $\mathcal{M}_1$ by $\mathcal{M}_2$ hold.

The refinement theorem enables Nickel to check noninterference for an implementation by checking the unwinding conditions for the interface specification and the refinement conditions (see §4). As with the unwinding theorem, the state invariants $\mathcal{I}_1$ and $\mathcal{I}_2$, the observational equivalence relation $\approx$, and the data refinement relation $\propto$ are *untrusted* for establishing noninterference.

3.5 Discussion and limitations

Nickel's formulation of noninterference falls into the category of *intransitive noninterference* [67]; in other words, it allows the can-flow-to relation of a policy to be either transitive or intransitive. As explained in §3.1, this flexibility is particularly useful for verifying practical systems, which often require downgrading operations. In addition, unlike classical noninterference, Nickel uses a state-dependent `dom` function, inspired by the formulation used to verify multiapplicative smart cards [68] and the seL4 kernel [57].

Nickel extends previous work in the following ways: the formulation supports a general set of policies and systems, which enables us to verify DIFC in NiStar (§6) and isolation in NiKOS and ARINC 653 (§7); all of its verification conditions for unwinding and refinement are expressible using an SMT solver, enabling automated verification to minimize the proof burden; and it provides a restricted form of refinement that preserves noninterference from an interface specification to an implementation.

Nickel's formulation of noninterference has the following limitations. It cannot uncover covert channels based on resources that are not captured in the interface specification, such as timing, sound, and energy. Modeling the effects of these resources is an orthogonal problem. Recent microarchitectural attacks [5, 42, 50] suggest the need for new hardware designs and primitives in order to eliminate such channels [21, 22].

Nickel does not support reasoning about concurrent systems. Concurrency is challenging not just for verification in general, but also for its implications on noninterference [71, 75]. In addition, Nickel models systems as deterministic state machines and requires developers to eliminate nondeterminism from the interface design (see §5). This requirement enables better proof automation and simplifies noninterference under refinement, but it restricts the types of interfaces that Nickel can verify [77].

Nickel's can-flow-to relation $\rightsquigarrow$ is state-independent, which means that Nickel cannot reason about dynamic, state-dependent policies [17] (though state-dependent `dom` functions partially compensate for this limitation). Moreover, Nickel's notion of refinement requires the interface specification and the implementation to use the same sets of actions and domains; this equality is sufficient for verifying systems like NiStar and NiKOS. Extending Nickel to support dynamic policies and more flexible refinements [76] would be useful future work.

4 Using Nickel

This section explains how the Nickel framework works and describes the steps needed to design and verify information flow control systems using Nickel.

Figure 7 depicts an overview of the Nickel framework and the required inputs from system developers (shaded

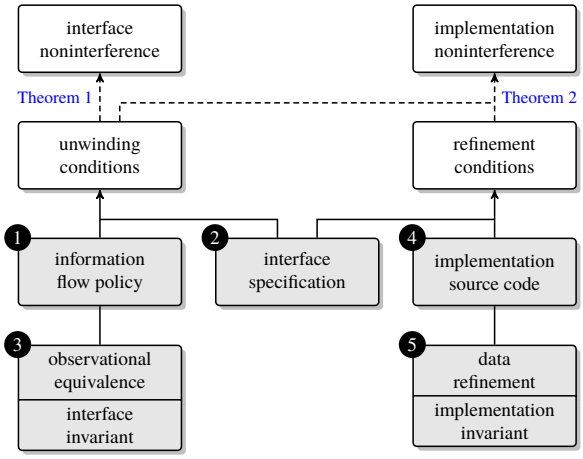


Figure 7: An overview of development flow using Nickel. Shaded boxes denote files written by system developers and the rest are provided by the framework. Circled numbers denote the steps. Solid and dashed arrows denote proof flows in SMT and Coq, respectively.

boxes with circled numbers). As part of the framework, the unwinding and refinement theorems (Theorem 1 and Theorem 2) serve as the metatheory for Nickel. We have formalized and proved both theorems using the Coq interactive theorem prover [74].

Developers write the system implementation in C and specify the rest of the inputs in Python. In particular, the development flow of using Nickel is the following:

1. Write the intended information flow policy to serve as the top-level specification of the system.
2. Model the system as a state machine and write a precise specification of each operation in the interface.
3. Construct a state invariant and observational equivalence for the interface specification, and invoke Nickel to check the unwinding conditions.
4. Implement each operation in the interface.
5. Construct a state invariant for the implementation and data refinement between the interface specification and the implementation, and invoke Nickel to check the refinement conditions.

Nickel extends the specification and verification infrastructure from Hyperkernel [62] to support reasoning about noninterference. It reduces all the inputs to SMT constraints—for instance, by performing symbolic execution on the LLVM intermediate representation of the implementation—and invokes Z3 to verify noninterference by checking the unwinding and refinement conditions. As with Hyperkernel, the initialization and glue code of the implementation is unverified. Interested readers can refer to Nelson et al. [62] for more information.

For verifying noninterference for an interface specification, the trusted computing base includes the information flow policy, the checker of unwinding conditions from Nickel, and Z3. For verifying noninterference for an implementation, it further includes the checker of refinement conditions from Nickel and the unverified initialization and glue code of the implementation.

Below we highlight two features of the development flow using Nickel.

A simple API for specifying the policy. As described in §3.1, a policy consists of a set of domains, a can-flow-to relation over domains, and a `dom` function associating each action in a state with a domain. Nickel provides a simple and intuitive API for specifying policies.

As an example, recall the isolation policy in Figure 3: each process p_i is a domain; the permitted flows in the system are: $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$ for $i \in [0, n-1]$. In Nickel, this policy is written as follows:

```
class ProcessDomain:
    def __init__(self, pid):
        self.pid = pid

    def can_flow_to(self, other):
        # Or is a built-in logical operator
        return Or(
            self.pid == 0,          # p0 ~> pi
            other.pid == 0,         # pi ~> p0
            self.pid == other.pid, # pi ~> pi
        )
```

In addition, the `dom` function of this policy returns the process currently running by default, or the scheduler p_0 for context switching actions (say, the `yield` system call):

```
class State:
    current = PidT() # PidT is an integer type
    ...

def dom(action, state):
    if action.name == 'yield':
       return ProcessDomain(0)
    else:
       return ProcessDomain(state.current)
```

This is all Nickel needs for the policy of NiKOS (§7).

Since a policy is the top-level specification of a system and must be trusted, developers should carefully audit the policy and ensure that it captures the design intention. We hope that the simple API for policies provided by Nickel makes auditing easier.

Debugging through counterexamples. To verify noninterference for an interface specification, Nickel checks the unwinding conditions from Theorem 1. If verification fails, Nickel produces a counterexample that illustrates the violation, including the operation name, an assignment of the operation arguments and system state(s), and the offending unwinding conditions.

Counterexamples provide useful information for debugging two types of failures. First, the violation may be in the interface specification, indicating a covert channel. Developers can use the counterexample to understand the violation and iterate on the interface design (see §5 for guidelines) until verification passes. Second, the state

invariant or the observational equivalence may be insufficient to establish noninterference. Developers can consult the counterexample to fix these inputs. Debugging the verification of an implementation follows similar steps.

5 Designing interfaces for noninterference

We have applied Nickel to verify noninterference in three systems: NiStar (§6), NiKOS (§7), and ARINC 653 (§7). While they have different information flow policies, our experience with these systems suggests several common guidelines for interface design.

Perform flow checks early. In general, operations need to validate parameters, especially those from untrusted sources (e.g., user-specified values in system calls), and return error codes indicating the cause of failure. As described in §2, returning error codes requires care to avoid covert channels. One simple way to avoid such channels is to use fewer error codes (or drop error codes altogether), but doing so makes debugging applications difficult.

NiStar addresses this issue by performing flow checks as early as possible. For example, many system calls need to check whether the current thread has permission to access specified data. After such a flow check succeeds, the system call has more liberty to validate parameters and return more specific error codes without violating noninterference.

Limit resource usage with quotas. Shared resources can lead to covert channels due to resource exhaustion. Systems may impose a quota on shared resources for each domain to avoid such channels. There are several quota schemes. One simple scheme is to statically assign predetermined quotas to domains; for instance, allowing processes to allocate only a predetermined number of identifiers for child processes [10]. However, this scheme limits the functionality of the system if the quota is too low, and wastes resources if the quota is set too high.

A more flexible and explicit quota scheme is to organize resources into a hierarchy of *containers* [4, 69, 82], where each container has a quota for resources such as memory and CPU time. A thread can allocate objects from a container, including creating subcontainers, if the container has sufficient quota and the policy allows the thread to access the container. A thread can also transfer quotas between two containers if the policy allows the thread to access both containers. NiStar uses containers to manage resources.

Partition names among domains. Resource names in a shared namespace, such as thread identifiers and page numbers, can lead to covert channels. A per-domain naming scheme partitions names among domains to eliminate such channels. A classical example is using ⟨process identifier, virtual page number⟩ pairs to refer to memory pages, effectively partitioning page numbers among processes. As another example, a system with container-based resource management may use ⟨container identifier, resource identifier⟩ pairs to refer to resources [82]; a thread may access the resource only if the policy permits it to access the container. Both NiStar and NiKOS employ per-domain naming schemes.

Encrypt names from a large space. Using encrypted names is an alternative way to address covert channels due to resource names. Many DIFC systems allocate sequential identifiers for resources, but return *encrypted* values to make them unpredictable [15, 45, 82]. This design technically violates noninterference, but since the identifier space is sufficiently large (e.g., 64 bits), the amount of information that can be leaked through this channel is negligible in practice. However, verifying noninterference for this design would require probabilistic reasoning [44] and complicate the semantics of noninterference [17: §6.4]. We therefore do not use encrypted names for the systems verified using Nickel.

Expose or enclose nondeterminism. As mentioned in §3.5, Nickel does not allow nondeterministic behavior in the interface specification (for instance, a system call that allocates an unspecified physical page), since doing so would complicate refinement for noninterference.

There are several options for revising the semantics of such system calls to eliminate nondeterminism. The first option is to make the (nondeterministic) decision explicit as a system call parameter, for example, asking user space to decide which page to allocate, similarly to exokernels [18, 37, 62]. The second option is to ask developers to explicitly describe the behavior (e.g., the allocation algorithm) as part of the interface specification. This makes the interface specification less abstract but simplifies the verification of noninterference under refinement; NiStar uses this option for memory management. The third option is to enclose the source of nondeterminism below the interface [28], for example, using virtual addresses to refer to memory pages and removing the use of physical pages from the interface. NiKOS uses this option.

Reduce flows to the scheduler. An OS scheduler is generally associated with a powerful domain, such as in Figure 3. The scheduler decides and updates which process to run, and other domains usually need to access this information (e.g., to look up the process currently running), creating inherent flows from the scheduler to other domains. Many scheduling approaches access information about processes to make scheduling decisions, creating flows from other domains to the scheduler. The combination of these flows makes the scheduler a powerful domain that two processes might exploit to communicate.

One way to control this risk is to enforce a stricter policy that prohibits flows *to* the scheduler. This policy restricts the power of the scheduler, since it can no longer query state that belongs to other domains. One simple design that satisfies this policy is to use a static, predetermined schedule [1, 57] that does not need to query the system state for scheduling decisions. NiStar instead satisfies this policy with a more flexible design: like exokernels [18, 37], it allows applications to allocate time slices to implement dynamic scheduling policies. Unlike exokernels, NiStar performs flow checks at run time to prevent these allocations introducing covert channels (see §6.2).

6 DIFC in NiStar

NiStar is a new OS kernel that supports decentralized information flow control (DIFC). NiStar's design is inspired by HiStar [82]: the kernel tracks information flow using labels and enforces DIFC through seven object types, and a user-space library implements POSIX abstractions on top of these kernel object types. Unlike HiStar, however, we have formalized NiStar's information flow policy and verified that both its interface specification and implementation satisfy noninterference for this policy. This section describes how we designed the NiStar interface to eliminate covert channels and used Nickel to achieve automated verification.

6.1 Labels

Like other DIFC systems [23, 45, 65], NiStar uses tags and labels to track information flow across the system. It follows a scheme used in DStar [83] and a revised version of HiStar [84]. A tag is an opaque integer, which has no inherent meaning. For instance, Alice uses tags t_S and t_I to represent the secrecy and integrity of her data, respectively. A label is a set of tags. Every object in the system is associated with a triple of ⟨secrecy, integrity, ownership⟩ labels, which we designate as the domain of the object. For instance, Alice labels her files with $\langle\{t_S\},\{t_I\},\varnothing\rangle$.

We use Figure 8 as an example to illustrate how Alice can constrain untrusted applications using labels. Suppose Alice launches a spellchecker to scan her files; the spellchecker consults a shared dictionary and prints the results (misspelled words) to her terminal. An updater periodically queries a server through the `netd` daemon and keeps the dictionary up to date. Alice trusts her `ttyd` daemon to declassify data only to her terminal. She trusts neither the spellchecker nor the updater, which may each be buggy, compromised, or malicious. Alice hopes to achieve the following security goals: (1) neither the spellchecker nor the updater can modify her files; and (2) her spellchecked files can not be leaked to the network.

Classical information flow control expresses policies using only secrecy and integrity labels (i.e., ignoring ownership). Given two objects with domains $L_1 = \langle S_1, I_1, O_1\rangle$ and $L_2 = \langle S_2, I_2, O_2\rangle$, respectively, it is safe in the classical model for information to flow from L_1 to L_2 if (1) the secrecy of S_1 is subsumed by that of S_2 and (2) the integrity of I_1 subsumes that of I_2: $S_1 \subseteq S_2 \wedge I_2 \subseteq I_1$. In other words, a flow is safe if it neither discloses secrets nor compromises the integrity of any object. For example, given the label assignment in Figure 8 and a system enforcing such flow checks, Alice can conclude that her files will not be modified by the spellchecker or the updater: her files have t_I in their integrity labels, but the spellchecker and updater do not, ruling out flows from them to her files.

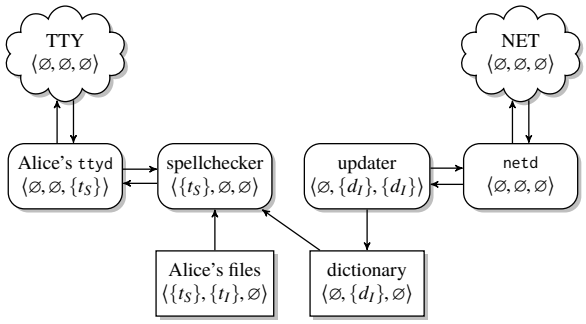


Figure 8: Information flow of a spellchecker and updater. Cloud boxes represent terminal (TTY) and network (NET); rounded boxes represent threads; and rectangular boxes represent data. Each object is associated with a triple of ⟨secrecy, integrity, ownership⟩ labels; arrows denote the flows of information allowed by these labels.

The classical model is often too restrictive for practical systems. For instance, a password checker needs to declassify whether login succeeds to untrusted users; as another example, to output misspelled words in Figure 8, the spellchecker (with t_S in secrecy) needs to communicate with to Alice's trusted `ttyd` (without t_S). Like other DIFC systems, NiStar supports such intentional downgrading without a centralized authority. It uses the ownership label to relax label checking for trusted threads, giving them the privilege to temporarily remove tags from secrecy labels (declassification) or add tags to integrity labels (endorsement), as follows:

Definition 3 (Safe Flow)**.** Information can flow from $L_1 = \langle S_1, I_1, O_1\rangle$ to $L_2 = \langle S_2, I_2, O_2\rangle$, denoted as $L_1 \rightsquigarrow L_2$, if and only if $(S_1 - O_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1 \cup O_1)$.

This can-flow-to relation is central to NiStar's information flow policy. $L_1 \rightsquigarrow L_2$ means that L_1 and L_2 can combine their ownership to allow the maximum flow from L_1 to L_2; that is, L_1 lowers its secrecy to $S_1 - O_1$ and raises its integrity to $I_1 \cup O_1$, while L_2 raises its secrecy to $S_2 \cup O_2$ and lowers its integrity to $I_2 - O_2$.

Referring to Figure 8, as information can flow from the spellchecker to Alice's `ttyd` given their label assignments, $\langle\{t_S\},\varnothing,\varnothing\rangle \rightsquigarrow \langle\varnothing,\varnothing,\{t_S\}\rangle$, Alice's `ttyd` is able to print out misspelled words. In addition, Alice can conclude that her files will not be leaked to the network: the

spellchecker cannot directly leak information to the network given its label assignment. The spellchecker can, however, indirectly write to Alice's terminal only through her `ttyd`, which she trusts to declassify data only to the terminal; no other threads in the system are trusted. This example shows how labels can minimize the amount of application code that must be trusted.

6.2 Kernel objects

NiStar provides seven object types:

- *labels* represent domains of objects;
- *containers* are basic units for managing resources;
- *threads* are basic execution units;
- *gates* provide protected control transfer;
- *page-table pages* organize virtual memory;
- *user pages* represent application data; and
- *quanta* represent time slices for scheduling.

Each object, other than labels, is associated with a domain of ⟨secrecy, integrity, ownership⟩ labels; only threads and gates can have non-empty ownership labels. The kernel interface consists of a total of 46 operations for manipulating these objects. Each operation performs flow checks among objects using their labels. NiStar's design goal is to ensure that the interface specification satisfies noninterference for the policy given by Definition 3.

NiStar largely follows HiStar's object types [82], with the following exceptions: it provides a new object type, quantum, for scheduling; and to make the interface finite and therefore amenable to automated verification, it uses fixed-sized page-table pages and user pages similar to Hyperkernel [62] and seL4 [40]. Interested readers can refer to Zeldovich et al. [84] for details of object types and label checks; below, we highlight three key differences in NiStar that close covert channels.

Given $L_1 = \langle S_1, I_1, O_1 \rangle$ and $L_2 = \langle S_2, I_2, O_2 \rangle$, we introduce the following notations for flow checks:

- $L_1 \sqsubseteq_R L_2$ means that L_1 *can be read by* L_2: $(S_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1)$.
- $L_1 \sqsubseteq_W L_2$ means that L_1 *can write to* L_2: $(S_1 - O_1 \subseteq S_2) \wedge (I_2 \subseteq I_1 \cup O_1)$.

As a shorthand, we write $L_2 \sqsubseteq_R L_1 \sqsubseteq_W L_2$ to mean that L_1 *can modify* L_2: $(L_2 \sqsubseteq_R L_1) \wedge (L_1 \sqsubseteq_W L_2)$. It is generally difficult for L_1 to modify L_2 without receiving any information in return (e.g., error code), and so this definition includes L_1 being able to read L_2. By definition, $L_1 \sqsubseteq_W L_3$ and $L_3 \sqsubseteq_R L_2$ together imply $L_1 \rightsquigarrow L_2$ for any L_1, L_2, and L_3; we will use this fact below to analyze covert channels. We denote $\mathcal{L}_x$ as the domain of object x.

Maintain accurate quotas in containers. Like HiStar, NiStar manages all system resources in a hierarchy of containers, starting from a root container created during kernel initialization. Each container maintains a set of quotas, indicating the amount of memory pages and time quanta it owns. A thread T may allocate an object O from a container C only if it can modify the container (i.e., $\mathcal{L}_C \sqsubseteq_R \mathcal{L}_T \sqsubseteq_W \mathcal{L}_C$), the new object does not exceed the authority of the thread (i.e., $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$), and the container has sufficient quota for the object.

NiStar maintains accurate quotas in containers, which differs from HiStar in two ways. First, NiStar sets the memory quota of the root container to be number of available physical pages upon booting, rather than infinity [82: §3.3], avoiding a potential covert channel due to resource exhaustion. Second, NiStar does not allow an object to be linked by multiple containers, which would require the kernel to conservatively charge each container as in HiStar. Instead, each object is uniquely owned by one container. This design leads to a simpler invariant: for each resource type, the sum of the quotas of each object in a container equals the total quota of the container.

Enforce can-write-to-object on deallocation. In HiStar, to deallocate an object O from a container C, a thread T must be able to write to the container, but not necessarily to the object itself. This relaxed check supports reclaiming *zombie* objects to which no one else can write (e.g., those with a unique integrity tag) [81]. However, it leads to a covert channel. Consider a thread T' whose domain permits it to read object O (i.e., $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$) but prohibits it from receiving information from thread T (i.e., $\mathcal{L}_T \not\rightsquigarrow \mathcal{L}_{T'}$). To bypass DIFC, thread T encodes a one-bit secret by either deallocating object O from container C or not. T' learns the secret by observing whether object O still exists [82: §3.2], violating noninterference since the label assignment prohibits information flow from T to T'.

NiStar enforces a stricter flow check on deallocation by requiring that thread T can write to object O (i.e., $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$). With this stricter check, this covert channel is closed: if thread T' can read object O (i.e., $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$), the new check implies that thread T' is permitted to receive information from thread T, since $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$ and $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$ together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$.

NiStar considers reclaiming zombie objects an administrative decision and leaves it to user space. Some systems may consider it legitimate for a user to create objects that no one else can reclaim; since NiStar enforces accurate quotas, adversarial users cannot create "runaway" zombie objects that exceed their quotas. On the other hand, a system wishing to reclaim zombie objects can emulate the HiStar behavior by setting up a trusted garbage collector with a powerful domain during booting, without baking this requirement into flow checks in the kernel.

Remove flows to the scheduler using quanta. As noted in §5, two processes can exploit the scheduler to communicate in violation of information flow policy. To close this channel, NiStar borrows the design of the exokernel scheduler [18] and extends it with label checking. NiStar associates the scheduler with domain $\langle \varnothing, \mathbb{U}, \varnothing \rangle$, where $\mathbb{U}$

denotes the universal label of all tags. This domain allows the scheduler to switch to any thread (its universal integrity allows it to influence any thread it runs) while restricting it from leaking information (its empty secrecy and ownership prevent it receiving secrets). The resulting scheduler allows applications to implement more flexible scheduling schemes compared to static scheduling.

NiStar introduces *time quanta* to allow the scheduler to make decisions while respecting this label assignment. The system is configured with a fixed number of quanta, each associated with a thread identifier for scheduling. Like other resources, all quanta are initially owned by the root container; a thread can move quanta between two containers only if it can modify both containers. To schedule thread T' at quantum Q, thread T writes the identifier of T' to Q. Thread T can perform this write only if it can write to quantum Q (i.e., $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_Q$).

To schedule using time quanta, assume that the system delivers an infinite stream of timer interrupts. Upon the arrival of a timer interrupt, the scheduler cycles through all the quanta in a round-robin fashion and retrieves the thread identifier T' associated with the next quantum Q. If quantum Q can be read by thread T' (i.e., $\mathcal{L}_Q \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$), the scheduler switches to T'; otherwise, it idles.

To see why these flow checks suffice to close the channel, suppose T is able to schedule T' to execute at quantum Q. The checks ensure $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_Q$ and $\mathcal{L}_Q \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$, which together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$; in other words, the label assignment permits T to communicate with T'.

This design closes covert channels arising from logical time. As mentioned in §3.5, physical timing is beyond the scope of this paper, for which NiStar provides no guarantees of noninterference.

6.3 Implementation

To demonstrate that NiStar's interface is practical, we have built a prototype implementation for x86-64 processors, and have applied Nickel to verify that both the interface specification and the implementation satisfy noninterference for the policy given by Definition 3.

To simplify verification, NiStar borrows ideas from previous verified OS kernels. First, like Hyperkernel [62], NiStar uses separate page tables for the kernel and user space. It uses an identity mapping for the kernel address space, sidestepping the complication of reasoning about virtual memory for kernel code [43]. Second, like seL4 [40], NiStar enables timer interrupts only in user space and disables them in the kernel. This restriction ensures that the execution of system calls and exception handling is atomic, avoiding reasoning about interleaved executions. Third, NiStar disables all other interrupts and requires device drivers to use polling, a common practice in high-assurance systems [1, 57].

For user space, we have ported the *musl* C standard library [59] to NiStar, running on top of an emulation layer for Linux system calls. A library implements the abstraction of Unix-like processes on top of NiStar's kernel object types, similar to HiStar's emulation layer [82]. The file-system service is implemented as a thin wrapper over containers and user pages, and the network service is provided by lwIP [13]. Although our current user space implementation is incomplete, it is able to run programs such as a set of POSIX utilities from Toybox, a web server, and the TinyEMU emulator to boot Linux.

7 Verifying isolation

Nickel generalizes to information flow control systems beyond DIFC. This section describes applying Nickel to two such systems: NiKOS and ARINC 653.

Process isolation. NiKOS is a small OS that enforces an isolation policy among processes (Figure 3). The interface of NiKOS mirrors that of a version of mCertiKOS as described by Costanzo et al. [10]. It consists of seven operations, including spawning a process, querying process status, printing to console, yielding, and handling a page fault. Like mCertiKOS, NiKOS imposes a memory quota on each process and statically partitions identifiers among processes, avoiding covert channels due to resource names and exhaustion (§5). We implemented a prototype of NiKOS for x86-64 processors and ported user-space applications from mCertiKOS. We used Nickel to verify that both the interface and implementation satisfy noninterference for the isolation policy. This effort took one author a total of two weeks.

We made one change to the design in order to verify noninterference. In mCertiKOS, the `spawn` system call creates a new process and loads an executable file; the specification of `spawn` models file loading as a no-op, whereas the implementation allocates pages and consumes memory quota [26]. In NiKOS, to match the memory quota in the specification with that in the implementation, `spawn` creates an empty address space and the page-fault handler lazily loads each page of the executable file instead.

Partition isolation. ARINC 653 [1] is an industrial standard for safety-critical avionics operating systems. It models the system as a set of *partitions* and defines an inter-partition communication interface comprising 14 operations. Figure 9 depicts its isolation policy among partitions: information can flow to a partition only from the *transmitter*, the scheduler, and itself. The transmitter forwards messages among partitions as configured at boot time; each dashed arrow represents a flow that can be independently enabled in the configuration. The scheduler uses a pre-configured fixed schedule, and so does not require flows from other domains to the scheduler (§5).

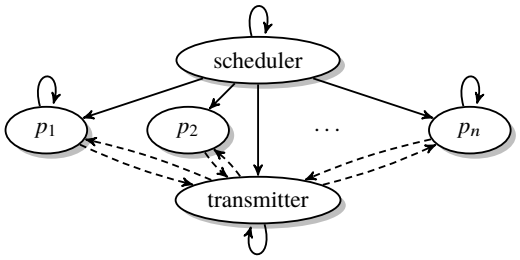


Figure 9: The isolation policy of ARINC 653: information can flow between the transmitter and each partition p_i for $i \in [1, n]$ as per a boot-time configuration (dashed arrows); it cannot flow between any two partitions, or from any partition or the transmitter to the scheduler.

Using Nickel, we formalized the specification of the communication interface based on the pseudocode provided by the ARINC 653 standard. Applying Nickel to verify noninterference for the partition isolation policy reproduced all three known covert channels first discovered by Zhao et al. [86], which were caused by missing partition permission checks, allocating identifiers in a shared namespace, and returning error codes that leak information; verification succeeded once we fixed these channels. This effort took one author a total of one week.

8 Experience

This section reports our experience with using Nickel and reflects lesson learned during development. Experiments ran on an Intel Core i7-7700K CPU at 4.5 GHz.

Covert channel discussion. To test the effectiveness of Nickel for detecting covert channels, we injected each of the examples in §2 into the NiStar interface specification. In each case, Nickel was able to find a counterexample pointing to the issue. As a concrete example, we switched NiStar's scheduler to a round-robin one. When verifying this round-robin scheduler, Nickel failed and produced a counterexample (§4).

Figure 10 shows empirical evidence of a covert channel by comparing the NiStar scheduler with the round-robin one. In this experiment, one process sampled the current (logical) time, while a background process repeatedly forked and then killed 30 child processes. The measuring process recorded the duration between scheduling points in terms of number of quanta. With the round-robin scheduler, the gaps observed by the measuring process vary as the background task forks and kills its children, creating patterns that indicate the covert channel. With the NiStar scheduler, which is verified using Nickel, the gaps between scheduling points remain constant regardless of the behavior of the background process. This result suggests that the Nickel is effective in identifying and proving the absence of covert channels.

Development effort using Nickel. Figure 11 shows the sizes of the three systems we verified using Nickel:

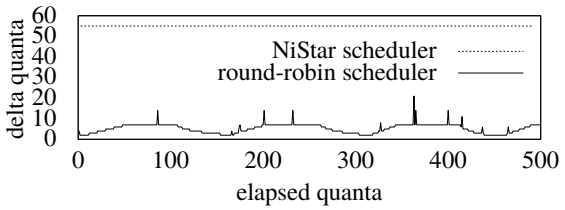


Figure 10: A round-robin scheduler leaks background thread behavior through patterns in logical time; no such pattern is observed in NiStar.

component	NiStar	NiKOS	ARINC 653
specification:			
information flow policy	26	14	33
interface specification	714	82	240
proof input:			
interface invariant	398	63	66
observational equivalence	127	56	80
implementation invariant	52	7	–
data refinement	139	30	–
implementation:			
interface implementation	3,155	343	–
user space implementation	9,348	389	–
common kernel infrastructure	4,829 (shared by NiStar/NiKOS)		

Figure 11: Lines of code for the three systems verified using Nickel.

NiStar, NiKOS, and ARINC 653. The lines of code for the interface implementations of both NiStar and NiKOS do not include common kernel infrastructure (C library functions and x86 initialization), and those of the user space implementations do not include third-party libraries (e.g., musl and lwIP). The implementation of the Nickel framework is split between the formalization of the metatheory (1,215 lines of Coq) and the verifier for the unwinding and refinement conditions (3,564 lines of C++ and Python).

The information flow policies for the three systems are concise compared to the rest of the specification and implementation, indicating the simplicity of creating policies ranging from DIFC to isolation using Nickel (§4).

In our experience, the most time-consuming part of the verification process was coming up with an appropriate observational equivalence relation—it was non-trivial to determine which part of the system state was observable by each domain, and the complexity increased as the size of the system state and the number of interface operations grew. We found the counterexamples produced by Nickel particularly useful for debugging and fixing observational equivalence. The specification and verification of NiStar, NiKOS, and ARINC 653 took one author six weeks, two weeks, and one week, respectively; as a comparison, implementing NiStar took several researchers roughly six months. This comparison shows that the proof effort required when using Nickel is low, thanks to its support for automated verification and counterexample generation.

Using Z3 4.6.0, verifying NiStar, NiKOS, and ARINC 653 on four cores took 72 minutes, 7 seconds, and 8 seconds, respectively.

Lessons learned. Our development of Nickel was guided by two motives. First, in our previous work on Hyperkernel [62], we proved memory isolation among processes, but this did not preclude covert channels through system calls; Nickel extends push-button verification to support proving stronger guarantees about noninterference. Second, we aimed to develop a general framework that can help analyze and design interfaces not only for isolation, but also for mechanisms as flexible as DIFC.

While designing Nickel, we spent a total of two months iterating through several formulations of noninterference before settling on the one described in §3. Among these alternatives were classical transitive noninterference [29] and intransitive noninterference [67], as well as variants such as nonleakage [56, 77]. As discussed in §3.5, Nickel's formulation has the advantage of supporting both a spectrum of policies and automated verification.

As Figure 7 shows, Nickel combines both automated and interactive theorem provers: Z3 automates proofs for individual systems, while the proofs in Coq improve confidence in Nickel's metatheory. Similar approaches have been used for the verification of compiler optimizations [72], static bug checkers [79], and Amazon's s2n TLS library [9]. We believe that this combination is an effective approach to developing verified systems.

9 Related work

Verifying noninterference in systems. Noninterference is a desirable security definition for operating systems looking to guarantee information flow properties [66]. For example, the seL4 microkernel [40] is proven to satisfy a variant of noninterference for a given access control policy [56, 57]; a version of mCertiKOS [27] includes a proof of process isolation [10]; Ironclad [32] proves end-to-end guarantees for applications using a form of input and output noninterference; and Komodo [19] proves noninterference for isolated execution of software-based enclaves. Noting the difficulty of extending noninterference proofs to concurrent systems, Covern [58] provides a logic for the shared memory setting. Noninterference also has applications in secure hardware [20, 21], programming languages [49, 73], as well as browsers and servers [36, 64]. Nickel takes inspiration from these efforts, focusing on formalizations and interface designs that are amenable to automated verification of noninterference.

DIFC operating systems. Information flow control was originally envisioned as a mechanism to enforce multilevel security in military systems [2, 3]. *Decentralized* information flow control (DIFC) additionally allows applications to declare new classifications [60, 61]. The design of NiStar was influenced by prior DIFC operating systems [7, 15, 45, 65, 82], particularly HiStar and Flume.

HiStar [82, 84] enforces DIFC with a small number of types of kernel objects. All label changes in HiStar are explicit, closing the covert channel in Asbestos due to implicit label changes [15]. NiStar's design draws from HiStar, using a similar set of kernel object types, but adapted to close remaining covert channels and enable automated verification.

Flume [45] is a DIFC system built on top of the Linux kernel. Building on top of an existing kernel makes porting easier, but expands Flume's TCB. Flume's design has a pen-and-paper proof [44] of noninterference for a single label assignment, modeled using Communicating Sequential Processes [34]; a more general formalization of Flume is given by Eggert [16]. NiStar takes this effort a step further, with the first noninterference proof of both the interface and implementation of a DIFC OS kernel.

Reasoning about information flows for applications. Assigning DIFC labels for applications is a non-trivial task. To help application developers, Asbestos offers a domain-specific language [14] for generating label assignments from high-level specifications. The SWIM tool [30] generates label assignments from lists of prohibited and allowed flows, and has been further extended using synthesis techniques [31]. These tools can benefit from a precise specification of the DIFC framework they use to implement policies for, such as the one provided by NiStar.

10 Conclusion

Nickel is a framework for designing and verifying information flow control systems through automated verification techniques. It focuses on helping developers eliminate covert channels from interface designs and provides a new formulation of noninterference to uncover covert channels or prove their absence using an SMT solver. We have applied Nickel to develop three systems, including NiStar, the first formally verified DIFC OS kernel. Our experience shows that the proof burden of using Nickel is low. We believe that Nickel offers a promising approach to the design and implementation of secure systems. All of Nickel's source code is publicly available at `https://unsat.cs.washington.edu/projects/nickel/`.

Acknowledgments

We thank Jon Howell, Kevin Zhao, the anonymous reviewers, and our shepherd, Raluca Ada Popa, for their feedback. We thank Nickolai Zeldovich and Ronghui Gu for answering our questions on HiStar and mCertiKOS, respectively. This work was supported in part by DARPA under contract FA8750-16-2-0032, by NSF under grant CCF-1651225, and by a Google Faculty Award.

References

[1] ARINC 653. Avionics application software standard interface: Part 1, required services, August 2015.

[2] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[3] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford, MA, April 1977.

[4] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, Baltimore, MD, August 2018.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[7] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 2012.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Chateau Lake Louise, Banff, Canada, October 2001.

[9] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of Amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, pages 430–446, Oxford, United Kingdom, July 2018.

[10] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, Santa Barbara, CA, June 2016.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, March–April 2008.

[12] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 (5):236–243, May 1976.

[13] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. Swedish Institute of Computer Science, February 2001.

[14] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM EuroSys Conference*, pages 301–313, Glasgow, Scotland, April 2008.

[15] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, Brighton, United Kingdom, October 2005.

[16] Sebastian Eggert. *Security via Noninterference: Analyzing Information Flows*. PhD thesis, Kiel University, July 2014.

[17] Sebastian Eggert and Ron van der Meyden. Dynamic intransitive noninterference revisited. *Formal Aspects of Computing*, 29(6):1087–1120, June 2017.

[18] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, December 1995.

[19] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, October 2017.

[20] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 555–568, Xi'an, China, April 2017.

[21] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[22] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, Jeju Island, South Korea, August 2018.

[23] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.

[25] John Graham-Cumming and J. W. Sanders. On the refinement of non-interference. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 35–42, Franconia, NH, June 1991.

[26] Ronghui Gu. Private communication, August 2018.

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, November 2016.

[28] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–208, San Diego, CA, December 2008.

[29] J. T. Haigh and W. D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[30] William R. Harris, Somesh Jha, and Thomas Reps. DIFC programs by automatic instrumentation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 284–296, Chicago, IL, October 2010.

[31] William R. Harris, Somesh Jha, Thomas Reps, and Sanjit A. Seshia. Program synthesis for interactive-security systems. *Formal Methods in System Design*, 51(2):362–394, November 2017.

[32] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, October 2014.

[33] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972.

[34] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[35] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 143–157, San Francisco, CA, May 2012.

[36] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Security Symposium*, pages 113–128, Bellevue, WA, August 2012.

[37] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In

Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), pages 52–65, Saint-Malo, France, October 1997.

[38] Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[39] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, pages 109–118, Las Vegas, NV, December 2002.

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[41] Gerwin Klein, Thomas Sewell, and Simon Winwood. Refinement in the formal verification of the seL4 microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 323–339. Springer, January 2010.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[43] Rafal Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, University of New South Wales, July 2011.

[44] Maxwell Krohn and Eran Tromer. Noninterference for a practical DIFC-based operating system. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 61–76, Oakland, CA, May 2009.

[45] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, Stevenson, WA, October 2007.

[46] Leslie Lamport. Computation and state machines, April 2008.

[47] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[48] Butler W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, Bretton Woods, NH, October 1983.

[49] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 158–170, Long Beach, CA, January 2005.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, August 2018.

[51] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. *ACM Transactions on Storage*, 10 (1):31–44, January 2014.

[52] Heiko Mantel. Preserving information flow properties under refinement. In *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, pages 78–91, Oakland, CA, May 2001.

[53] John McLean. Security models and information flow. In *Proceedings of the 11th IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, May 1990.

[54] MITRE. CWE-209: Information exposure through an error message. https://cwe.mitre.org/data/definitions/209.html, January 2018.

[55] Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. https://www.openssl.org/~bodo/tls-cbc.txt, September 2014.

[56] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP)*, pages 126–142, Kyoto, Japan, December 2012.

[57] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow

enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[58] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *Proceedings of the 3rd IEEE European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018.

[59] musl. https://www.musl-libc.org/, 2018.

[60] Andrew Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–147, Saint-Malo, France, October 1997.

[61] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[62] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, Shanghai, China, October 2017.

[63] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, CA, March 2011.

[64] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 452–462, Edinburgh, United Kingdom, June 2014.

[65] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[66] John Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–21, Pacific Grove, CA, December 1981.

[67] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, December 1992.

[68] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart cards. In *Proceedings of the 6th European Symposium on Research in Computer Security*, pages 17–36, Toulouse, France, October 2000.

[69] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, December 1999.

[70] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[71] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 201–214, Copenhagen, Denmark, September 2012.

[72] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 111–121, Toronto, Canada, June 2010.

[73] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*, pages 352–367, London, United Kingdom, September 2005.

[74] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*, April 2018. URL https://doi.org/10.5281/zenodo.1219885.

[75] Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow

in Haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 187–202, Venice, Italy, July 2007.

[76] Ron van der Meyden. Architectural refinement and notions of intransitive noninterference. *Formal Aspects of Computing*, 24(4–6):769–792, July 2012.

[77] David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *Proceedings of the 9th European Symposium on Research in Computer Security*, pages 225–243, Sophia Antipolis, France, September 2004.

[78] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, Farmington, PA, November 2013.

[79] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. SpaceSearch: A library for building and verifying solver-aided tools. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Oxford, United Kingdom, September 2017.

[80] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, Cape Breton, Canada, June 2001.

[81] Nickolai Zeldovich. Private communication, April 2018.

[82] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, November 2006.

[83] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, San Francisco, CA, April 2008.

[84] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011.

[85] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th USENIX Security Symposium*, pages 17–32, Montreal, Canada, August 2009.

[86] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. Reasoning about information flow security of separation kernels with channel-based communication. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 791–810, Eindhoven, The Netherlands, April 2016.

Verifying concurrent software using movers in CSPEC

Tej Chajed, M. Frans Kaashoek, Butler Lampson[†], and Nickolai Zeldovich
MIT CSAIL and [†]Microsoft Research

Abstract

Writing concurrent systems software is error-prone, because multiple processes or threads can interleave in many ways, and it is easy to forget about a subtle corner case. This paper introduces CSPEC, a framework for formal verification of concurrent software, which ensures that no corner cases are missed. The key challenge is to reduce the number of interleavings that developers must consider. CSPEC uses mover types to re-order commutative operations so that usually it's enough to reason about only sequential executions rather than all possible interleavings. CSPEC also makes proofs easier by making them modular using layers, and by providing a library of reusable proof patterns. To evaluate CSPEC, we implemented and proved the correctness of CMAIL, a simple concurrent Maildir-like mail server that speaks SMTP and POP3. The results demonstrate that CSPEC's movers and patterns allow reasoning about sophisticated concurrency styles in CMAIL.

1 Introduction

Achieving high performance on a single computer requires concurrency, such as running on multiple cores or interleaving disk and network I/O with computation. Concurrent software, however, is difficult to get right because threads can interleave in many ways, and reasoning about all possible interleavings is hard. Furthermore, testing is insufficient, because there are usually too many interleavings to consider, and because it is difficult to reproduce a bug unless the developer knows the precise interleaving that caused it. By contrast, formal verification can prove that a system behaves correctly (i.e., satisfies its specification) in every possible interleaving, including all corner cases.

There has been some prior work on machine-checked verification of concurrent systems software on a single computer. For example, CertiKOS has verified spinlocks for protecting scheduling queues [13, 21]. As we discuss in detail in §2, that work focuses on lock-based concurrency. Systems in which concurrency takes the form of multiple processes sharing a file system tend to avoid the use of locks because they interact badly with crashes. This requires reasoning about many possible interleavings, since there is no lock enforcing sequential execution during critical sections. Work on a concurrent garbage collector [17, 18] supports reasoning about lock-free shared-memory concurrency, but relies on pen-and-paper proofs for key theorems, and does not support important proof patterns needed for CMAIL.

This paper presents CSPEC, a framework for specifying, implementing, and proving the correctness of concurrent systems. CSPEC supports reasoning about concurrent processes that share a file system, as well as about concurrent threads that share data structures in memory. All of CSPEC is implemented and proven in the Coq proof assistant [36].

To show that CSPEC makes it fairly easy to prove the correctness of concurrent software, we used it to develop a simple concurrent mail server, CMAIL. Typical mail servers such as Maildir do not use file locks for mail delivery, since locks are fragile if a process is killed or suspended while holding the lock [3]. Instead, Maildir relies on careful reasoning about atomicity and ordering of file system operations (e.g., writing data to a temporary file before renaming it into the user's mailbox directory). Mail delivery must interact safely with mail pickup (e.g., retrieving mail via POP3)—for instance, retrieving mail from a mailbox in the presence of concurrent deliveries to the same mailbox. Finally, other parts of the mail server do use POSIX file locking—for example, to ensure that a message cannot be retrieved and deleted at the same time.

The key challenge in CSPEC is to reduce the number of interleavings that the developer must consider in code like CMAIL's lock-free delivery. To achieve this, CSPEC uses the notion of *mover types* [27], which exploits the fact that certain operations are left- or right-commutative with respect to concurrent operations by other processes. CSPEC uses mover types to re-order operations so that processes appear to execute longer blocks of sequential code. This reduces the problem of reasoning about all interleavings to reasoning about just the atomic execution of these longer sequential blocks. CSPEC builds on prior work that used mover types to reason about concurrency [11, 16, 18], and provides the first general mover framework with a machine-checked proof of its implementation (see §2).

CSPEC allows the developer to separately tackle different aspects of the design by structuring the overall system as a stack of layers. Each layer has a formal specification and its own implementation and proof. CSPEC provides a library of patterns for different kinds of proofs that a layer might need, such as mover types, retry loops, abstracting state, partitioning state, and proving that the code follows

a protocol (i.e., a set of rules), such as accessing memory only while holding a lock.

We evaluate two key aspects of CSPEC: whether CSPEC makes it possible to do correctness proofs for sophisticated concurrent software, and whether the resulting concurrency translates into actual speedup. CMAIL is our primary case study of verifying concurrent software. CSPEC allowed us to handle its challenging concurrency patterns, such as a delivery process that modifies a mailbox directory at the same time the user is picking up mail, multiple delivery processes that write messages into the same mailbox, and concurrent processes sharing the same temporary directory to store partially received messages.

CSPEC's layering allowed us to decompose the overall correctness argument for CMAIL into smaller steps, each layer addressing a specific aspect of CMAIL's concurrency and using a CSPEC proof pattern to formally verify it. All of CSPEC's proof patterns were important in CMAIL, and most patterns were used multiple times. Designing and building CSPEC and CMAIL took two people approximately 6 months, on top of another 12 months of experimenting with several failed alternative designs. Experiments show that CMAIL's concurrency makes it run faster on a multi-core machine.

CMAIL's concurrency model is based on processes sharing a file system. CSPEC also allows developers to reason about other concurrency models. To demonstrate this, we specified a model of x86-TSO [34], consisting of a shared memory with per-core write buffers. On top of this model, we implemented and proved the correctness of an atomic counter. We used 10 layers to verify this counter, re-using proof patterns that we developed for CMAIL.

To summarize, the contributions of this paper are:

- CSPEC, a framework for verifying concurrent systems using mover types, which is fully machine-checked in Coq.
- A modular approach that simplifies proofs using layers and a library of proof patterns.
- An evaluation that uses CSPEC to formally prove the correctness of a concurrent mail server on top of a POSIX file system, and an atomic counter on top of a weak shared-memory model. The results demonstrate that CSPEC allows reasoning about a wide range of concurrency styles.

The source code of CSPEC and the example applications are publicly available at https://github.com/mit-pdos/cspec. Our prototype has several limitations. CMAIL does not include verified parsing or protocol implementations of SMTP or POP3. CSPEC uses Coq's extraction to generate executable code, which means the executable programs rely on either Haskell or OCaml at runtime; hence, one of these is part of the trusted computing base. Finally, CSPEC cannot be applied to existing software, since it requires the program to be written in CSPEC's framework.

2 Related work

CSPEC adopts many ideas from previous research in specification and verification of concurrent shared-memory and distributed-systems software.

Verification approaches. There are many ways to verify concurrent software. After experimenting with several different approaches (including several versions of concurrent separation logic [5] and rely-guarantee [12, 20]), we settled on using the state machines and refinement that underlie TLA and I/O automata [23, 24, 30, 31], combined with the proof pattern of movers [27].

CIVL [17, 18] (and its predecessor QED [11]) is the work most closely related to CSPEC, and CSPEC borrows many ideas from it. CIVL uses the state-machine approach with support for atomic actions, movers, a mover pattern inspired by CIVL's yield sufficiency automaton, and location invariants to reduce the proof burden. It is implemented as an extension to Boogie, and the authors used it to specify and verify a concurrent garbage collector that uses an algorithm by Dijkstra et al. [10] that has tricky concurrency reasoning. Subsequent work used CIVL to reason about concurrent programs on x86-TSO [4].

CSPEC borrows atomic actions and movers from CIVL, but differs in two ways. First, many of CIVL's proofs (e.g., all the proofs in §4 of [18]) are done with pen and paper [15], whereas all parts of CSPEC are machine-checked in Coq. Second, CSPEC supports some patterns not found in CIVL, such as retry loops, which were important for reasoning about concurrency in CMAIL. Furthermore, this paper reports on our experience in using CSPEC for a different application (namely, a file-system-based mail server rather than a concurrent garbage collector), which exhibits different styles of concurrency.

The advantage of the fact that CSPEC has machine-checked proofs, compared to CIVL's pen-and-paper proofs, is that it gives us confidence that all of the proof patterns are correct (once we prove them). This, in turn, makes it easier to experiment with proof patterns. For example, during the development of CSPEC, we added (and modified) a number of proof patterns (see §6). Having machine-checked proofs gave us confidence that we did not introduce any bugs.

Like CSPEC, CertiKOS's CCAL [14] organizes reasoning about concurrent execution into layers and has a linking theorem to "compile" top-level operations into bottom-layer operations. CCAL has been used for fully machine-checked proofs of several lock implementations and of CertiKOS's concurrent scheduling queue [13, 21].

CCAL has no notion of movers; it uses rely-guarantee-style reasoning to prove atomicity for operations in a shared log. The only case in which CCAL can avoid reasoning about interleaving is when a thread accesses only thread-private memory. This is insufficient for CMAIL, which accesses shared files and directories all the time: for instance, mail pickup can read a message that was just written by a concurrent delivery process.

Another notable example of verifying concurrent systems software is Microsoft's HyperV verification, which used VCC [7–9], but the work on VCC and HyperV appears to have stopped after verifying about 20% of HyperV [7]. In contrast to CSPEC, the VCC approach did not use mover types for reasoning about concurrency.

Distributed systems. Related work in verifying distributed systems focuses on network protocols (message loss and re-ordering) as well as node failures and network partitions, while assuming a static partitioning of state across nodes [16, 26, 33, 37]. The focus of CSPEC, in contrast, is on dynamic sharing of state between processes on a single node, and on the patterns that help developers construct proofs for different styles of concurrency. CSPEC does not address node failures.

IronFleet [16] uses the notion of trace inclusion and movers in their reduction argument, which has been machine-checked [19]. However, IronFleet's verified reduction argument is specialized for IronFleet's specific use case, and has a hard-coded list of movers: sending and receiving UDP packets, and acquiring and releasing locks [28]. In contrast, CSPEC is a general-purpose mover framework.

Mail servers. Affeldt and Kobayashi verified a part of a mail server written in Java, by manually translating the Java program into a Coq function, and verifying properties of the Coq function [1, 2]. They verified the SMTP receiver part of the mail server, but do not model the interaction between the mail server and the file system. We use CSPEC to verify CMAIL, which includes both delivery via SMTP as well as pickup via POP3, and prove that CMAIL correctly uses the file system.

Ntzik [32] developed a concurrent specification for POSIX file systems using a concurrent separation logic, and used it to reason about snippets of mail server code for spam filtering. In contrast, CMAIL is a fully operational concurrent mail server, with a complete specification and machine-checked proof of its implementation.

3 Goal and challenges

The goal of CSPEC is to allow developers to write specifications for concurrent systems software such as the mail server and to prove that an implementation satisfies the spec. The proof should ensure that every possible interleaving, no matter how unlikely, is handled correctly.

```
def deliver(user, msg):
  tmpname = "/tmp/%d" % getpid()
  f = open(tmpname, "w")
  f.write(msg)
  f.close()

  while True:
    mboxfn = "/var/mail/%s/%d" % (user, random())
    if link(tmpname, mboxfn) == ok:
      unlink(tmpname)
      return
```

Figure 1: Pseudocode for delivery in a Maildir-like mail server.

```
def pickup(user):
  files = readdir("/var/mail/%s" % user)
  messages = []
  for fn in files:
    f = open("/var/mail/%s/%s" % (user, fn))
    messages.append(f.read())
    f.close()
  return messages
```

Figure 2: Pseudocode for pickup in a Maildir-like mail server.

To illustrate why this is hard, consider a mail server running on top of a file system, as a prototypical example of concurrent systems software. A mail server performs two main operations: `deliver`, which accepts incoming messages and writes them to the file system, and `pickup`, which allows users to download their messages. A mail server typically runs many processes, which concurrently perform deliveries and pickups.

For instance, consider the Maildir-like [3] server shown in Figures 1 and 2. In Maildir, each user's mailbox is a directory containing one file for each message. Maildir does not use locks for most concurrency control; instead, `deliver` and `pickup` choose file names and issue file system operations that are carefully designed to avoid races.

`deliver` first writes the incoming message into a temporary file with a unique filename (based on the process ID) and then links the file into the user's mailbox directory with a randomly chosen name. If the link fails because the filename already exists (which can happen because of another delivery that chose the same random name), `deliver` retries it with a different filename. To read a user's messages, `pickup` calls `readdir` to list the files in the user's mailbox, and reads them one at a time.

Even though the code appears to be simple, it is carefully designed to handle many subtle interleavings of file system operations that arise when multiple concurrent processes invoke `deliver` and `pickup`. For example, `pickup` will not return any partially written messages to the user, because `deliver` will link a message into the user's mailbox only after it has been fully written to a file. As another example, two `deliver` processes will not overwrite each other's messages, because they choose distinct filenames in the temporary directory, and because they use `link` to atomically place a message into the mailbox directory if and only if the filename does not already exist.

```
Definition message := string.

(* Defines a new type, [Op], representing operations, where
   running an [Op retT] returns a value of type [retT]. *)
Inductive Op : forall (retT : Type), Type :=
  (* one operation is [Deliver], which takes two arguments,
     [u] and [msg], and returns a [bool] *)
| Deliver : forall (u : user) (msg : message), Op bool
| Pickup : forall (u : user),  Op (list (msgid * message))
| CheckUser : forall (u : user), Op bool
| Delete : forall (u : user) (id : msgid), Op unit.

(* The abstract state is a two-level map: from users to
   mailboxes, which are maps from IDs to messages. *)
Definition State := Map.t user (Map.t msgid message).

(* The semantics, defining valid transitions for operations. *)
Inductive step :
  (* Transitions depend on the operation being executed, the
     current PID, and the initial state .. *)
  forall `(op : Op retT) (pid : nat) (st : State)
  (* .. and determine the operation's return value (whose
     type depends on the operation) and the final state *)
         (r : retT) (st' : State), Prop :=
| StepDeliverOK : forall u msg pid id st mbox,
  (* if user [u]'s mailbox is [mbox] *)
  Map.MapsTo u mbox st ->
  (* .. and message ID [id] is not used in [mbox] *)
  ~ Map.In id mbox ->
  (* .. then the following is a valid transition: *)
  (Deliver u msg, pid, st) |->
      (true, Map.set u (Map.set id msg mbox) st)
| StepDeliverErr : forall u msg pid st,
  (Deliver u msg, pid, st) |-> (false, st)
(* Some transitions omitted for space reasons *)
| StepDelete : forall u id pid st mbox,
  Map.MapsTo u mbox st ->
  (Delete u id, pid, st) |->
      (tt, Map.set u (Map.remove id mbox) st)
where "(op, pid, st) |-> (r, st')" := step op pid st r st'.
```

Figure 3: Specification of the mail server. Code snippets in this paper have been simplified for readability; the full code of CSPEC and CMAIL is available at https://github.com/mit-pdos/cspec.

4 Approach to proving atomicity

CSPEC's approach to verifying concurrent software is to specify the atomic semantics of operations such as `deliver` and `pickup`, and then prove that their implementations, such as the code shown in Figure 1 and Figure 2, meet their specs.

To use CSPEC, a developer first specifies the desired behavior of each operation if it were to execute atomically; then writes code in CSPEC to achieve this behavior, even when running concurrently; and finally the developer proves that the code indeed meets the atomic spec in all possible cases, with the help of CSPEC's proof patterns.

For example, Figure 3 shows the atomic spec of the main operations in CMAIL. The first statement in Figure 3 defines the set of allowed operations, using a Coq inductive type called `Op`. The next statement defines the abstract state. The last statement defines the semantics, by describing the allowed transitions using a Coq inductive type. For example, the first allowed transition, `StepDeliverOK`, states one legal way for a `Deliver` operation to execute with some arguments `u` and `msg`. Namely, if `u`'s mailbox is `mbox`, then `Deliver` adds the incoming message with a new identifier `id` in the user's mailbox. Here, `Deliver` denotes the primitive operation in the semantics, whereas

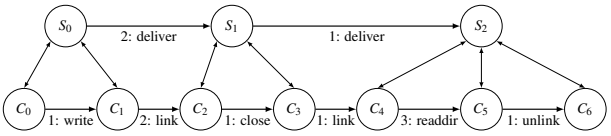


Figure 4: Example diagram of a simulation proof, connecting code from Figure 1 with the spec from Figure 3. In the example, processes 1 and 2 each deliver a message concurrently, while process 3 is running `pickup`.

the pseudocode of `deliver` from Figure 1 describes a possible implementation of `Deliver`.

To understand why proving correctness is hard, consider the approach based on a *simulation* proof [30], used by many frameworks [16, 25, 35]. The idea is to define an *abstraction relation* that connects the spec-level states with code-level states, and to show that this relation is preserved by every possible transition at the code level.

Figure 4 shows a simulation argument for one execution of the mail server: two processes concurrently delivering a mail message. At the bottom are code-level states, representing the states and transitions of the file system, corresponding to code from Figure 1 (in this example, the mail server is handling an incoming SMTP message). At the top are spec-level states, representing the abstract state and transitions of the mail server, corresponding to the specification in Figure 3. The abstraction relation, shown as vertical arrows, captures the correspondence between the abstract spec-level state (set of messages in each user mailbox) and the concrete code-level state (files and directories representing the mailboxes). For each code-level transition, the simulation proof shows that the new code-level state corresponds to a spec-level state after zero or more spec-level transitions.

Proving atomicity using simulation turns out to be hard, because it requires the developer to consider many possible interleavings, such as the one shown at the bottom of Figure 4 among others. This leads to a secondary complication: the abstraction relation must describe all reachable code-level states, including ones in which many processes are halfway through executing their updates.

We would like to reduce the problem of reasoning about concurrent execution to reasoning about sequential execution as much as possible. To provide some intuition for why this might work, consider `deliver` from Figure 1. We would like to ignore interleavings with other processes before `link` (lines 2-8) and after `link` (lines 9-11), because operations on lines 2-8 affect that process's temporary file, which is specific to that process's PID. Other processes will not interfere with it. However, the interleavings with respect to `link` (line 9) do matter, because the message created by `link` in the shared mailbox can now affect other processes running `deliver` or `pickup`.

To formalize this intuition, CSPEC uses the idea of left-, right-, and non-movers [27], which captures the notion that operations from different processes might (or might

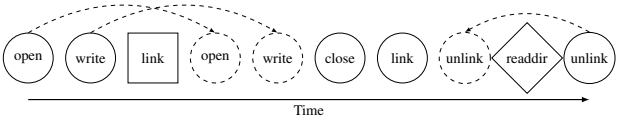


Figure 5: Example interleaving of file system operations executed by 3 separate processes: circles correspond to a process running `deliver`, squares correspond to another process running `deliver`, and diamonds correspond to another process running `pickup`. Dotted operations and arrows indicate re-ordering with the help of mover types.

not) commute with one another. Movers help CSPEC reason about atomicity, by proving that certain sets of interleavings all produce the same outcome, and hence that it suffices to consider just the interleaving where the code executes atomically.

Consider the interleaving in Figure 5, which depicts the code-level steps from the bottom of Figure 4. (Ignore the dashed elements for now.) In this example, the `open`, `write`, and `close` operations from process 1 (denoted by circles) are *right-movers*, which means that moving their execution to the right in the diagram (past the transitions of other processes) produces the same outcome. This is because `open` and `write` modify a temporary file that's named by the process ID and hence not accessed by any other process, and because `close` does not interact with other processes at all. Similarly, the `unlink` operation from process 1 is a *left-mover*, which means that it can be moved earlier (left) in the execution (past the transitions of other processes) without changing the outcome. However, note that the `link` operation from process 1 is neither a left- or right-mover (i.e., a non-mover), since moving it to the left or right can change the outcome by affecting a `readdir` from a concurrent `pickup`.

By using left- and right-movers in Figure 5, we can re-order the execution of `deliver` in process 1 to be atomic, as shown by the dashed elements in Figure 5. This re-ordering corresponds to a sequential execution of `deliver`, and allows us to prove that `deliver` can be thought of as executing atomically. We do the same style of reasoning for `pickup`, showing that we can rearrange operations so that they form a sequential execution of `pickup`, and then proving that the implementation preserves the atomicity of `pickup`. This further allows us to prove correctness of arbitrary interleavings of processes by considering just the sequential executions of `deliver` and `pickup`.

5 Design of CSPEC

This section provides an overview of CSPEC's design by describing what a layer is, how CSPEC defines correctness, and how a developer proves an implementation correct.

5.1 Layers

CSPEC's workflow involves defining *layers*. The spec of a layer has three parts: the set of operations (`Op`), the state manipulated by those operations (`State`), and the

```
Definition pathname := list string.

Inductive Op : forall (retT : Type), Type :=
| Read : forall (pn : pathname), Op (option string)
| Link : forall (src : pathname) (dst : pathname), Op bool
| Unlink : forall (pn : pathname), Op unit
| List : forall (dirpn : pathname), Op (list string)
(* Some operations omitted for space reasons *).

Inductive State : Type :=
| ST : forall (Files : Map.t pathname string)
              (Locks : Map.t pathname bool), State.

Inductive step : forall `(op : Op retT) (pid : nat)
  (st : State) (r : retT) (st' : State), Prop :=
| StepReadOK : forall pn fs pid msg locks,
  Map.MapsTo pn msg fs ->
  (Read pn, pid, ST fs locks) |-> (Some msg, ST fs locks)
| StepReadNone : forall pn fs pid locks,
  ~ Map.In pn fs ->
  (Read pn, pid, ST fs locks) |-> (None, ST fs locks)
(* Some transitions omitted for space reasons *)
| StepLinkOK : forall fs pid dst data pn locks,
  Map.MapsTo pn data fs ->
  ~ Map.In dst fs ->
  (Link pn dst, pid, ST fs locks) |->
      (true, ST (Map.set dst data fs) locks)
| StepLinkErr : forall fs pid dst pn locks,
  (Link pn dst, pid, ST fs locks) |-> (false, ST fs locks)
where "(op, pid, st) |-> (r, st')" := step op pid st r st'.
```

Figure 6: Low layer for the mail server example.

semantics, describing how each operation updates this state and what value it returns (the `step` relation).

For example, the top layer of the mail server is the spec shown in Figure 3. The bottom layer is the file system, partially shown in Figure 6. This layer defines the file system operations, the file system state (a tuple, called `ST`, consisting of a map representing the contents of all files, and a map representing whether each file is locked using POSIX file locking), and the results of each operation: how it updates the state and what values it returns.

Layers are an important modularity technique. Many proofs in CSPEC require considering all possible transitions made by other processes (e.g., when proving that an operation is a right- or left-mover). Doing so directly on top of the file system layer would be tedious, because there are many possible transitions (corresponding to many operations), and because the transitions operate in terms of low-level file system state. Re-defining the operations and state in an intermediate layer can simplify the proof, because the state is smaller and there are fewer operations to consider. For instance, to prove CMAIL, we decomposed it into 13 layers as shown in Figure 7, with each layer (except the bottom) implemented using the operations of the layer below it. As an example, Figure 8 shows the implementation connecting the MailboxTmpAbs and Deliver layers.

Connecting two layers requires writing code for every higher-level operation that uses only lower-level operations, and a proof that this code meets the layer's spec. CSPEC then links multiple layers together by chaining their implementations and proofs. It's much easier to do the proofs if they map onto CSPEC's proof patterns.

	Layer name	Operations	State	Pattern
Per-user layers	MailServerComposed	Deliver, Pickup, Delete, CheckUser (Figure 3)	Messages in user mailboxes (Figure 3)	Part
	MailServerPerUser	Per-user Deliver, Pickup, Delete	Messages in one user's mailbox	Abs
	MailServerLockAbs	*same as above*	+ Lock on mailbox for serializing Pickup and Delete	Mov+Prot
	Mailbox	+ List and Read; - Pickup	*same as above*	Abs
	MailboxTmpAbs	*same as above*	Additional temporary directory	Mov
	Deliver	+ Create, Link, and Unlink; - Deliver	*same as above*	Mov+Prot
	DeliverListPid	+ Filtered List returning files with caller's PID	*same as above*	Mov+Prot
	MailFS	+ GetPID; - Filtered List	*same as above*	Abs
	MailFSStringAbs	*same as above*	File names are strings instead of pairs	Mov
	MailFSString	Operations now in terms of string names	*same as above*	Abs
	MailFSPathAbs	*same as above*	Per-user file system	Mov
	MailFSPath	Per-user file system operations	*same as above*	Part+Abs
	MailFSMerged	File system operations (Figure 6)	File system (Figure 6)	

Figure 7: Layers used for verifying the mail server. The operations column describes the `Op` type for that layer. The state column describes the abstract state, `State`, over which the layer's semantics are defined. The pattern column lists the CSPEC proof patterns (described in §6) used for connecting two layers. This layering corresponds to "plan 1" described later on in §8.1; not shown is one intermediate layer used for "plan 2."

For example, some of the lower layers in Figure 7 deal with how the mail server state is encoded using directories and file names. That is, these layers have a different definition of `State`, but typically the same list of operations (i.e., same `Op`) as higher layers. All of the layers above, however, assume that the mail server's mailbox is completely disjoint from the temporary directory, and assume that file names are pairs of process ID and message ID (i.e., their `State` is just a map, as in Figure 3). As a result, the code and proofs at higher layers need not worry about file name encoding, pathnames, traversing directories, etc.

```
Definition deliver_core (msg : message) :=
  ok <- Call (DeliverOp.CreateWriteTmp msg);
  match ok with
  | true =>  ok <- Call (DeliverOp.LinkMail);
             _  <- Call (DeliverOp.UnlinkTmp);
             Ret ok
  | false => _ <- Call (DeliverOp.UnlinkTmp);
             Ret false
  end.

Definition compile_op '(op : MailboxOp.Op T) :=
  match op with
  | MailboxOp.Deliver msg => deliver_core msg
  | MailboxOp.Read pn     => Call (DeliverOp.Read pn)
  | MailboxOp.Delete pn   => Call (DeliverOp.Delete pn)
  ...
  end.
```

Figure 8: Implementation connecting the MailboxTmpAbs and Deliver layers.

5.2 Defining correctness

CSPEC's definition of correctness revolves around the observable behaviors allowed by the specification, and the observable behaviors that can be produced by the implementation. CSPEC uses a standard notion of correctness: it requires that the behaviors of the implementation be a subset of the behaviors allowed by the spec.

More formally, CSPEC models the interaction with the outside world using the notion of *events* [24, 30]. The idea is to annotate operations that interact with the outside world (e.g., accepting a connection, reading or writing network messages, closing a connection, etc) as producing *events*. These events reflect the external behavior of our system: SMTP messages coming in and being acknowledged, and POP3 requests coming in and getting responses. The sequence of events produced by a system thus defines its externally visible behavior, which we call a *trace*.

CSPEC defines correctness of an application by requiring that the traces of events produced by the application when using the concurrent implementation of operations (e.g., `deliver` and `pickup`) must be a subset of the traces that can be produced by the application using the specification of those operations (e.g., Figure 3 for the mail server). In other words, if the actual implementation of the system can exhibit some behavior, then this behavior must be allowed by the atomic specification.

Trace inclusion is a good fit for specifying concurrent systems, compared to some of the alternative approaches that have been used by recent systems, such as postconditions [6]. Postconditions allow specifying the return values from a procedure, but this does not help with procedures that never return, such as the mail server that accepts incoming connections in an infinite loop.

CSPEC also uses the notion of trace inclusion to define the correctness of intermediate layers, such as the 13 layers used in CMAIL. Transitively, if each layer produces a subset of traces allowed by the layer above it, the entire stack of layers is correct: the traces produced by the bottom-most code are a subset of traces allowed by the top-most specification.

5.3 Implementation

An implementation is a module that provides one function, `compile_op`, which implements higher-level operations in terms of lower-level operations. For instance, CMAIL has 12 such implementations, connecting its 13 layers. Implementations of multiple layers can be chained together; for instance, CMAIL chains together its implementations to translate high-level operations like `Deliver` and `Pickup`

```
Lemma createwritetmp_right_mover : forall data,
  right_mover DeliverRestrictedAPI.step
              (DeliverOp.CreateWriteTmp data).
Proof.
  unfold right_mover; intros.
  ...
Qed.

Lemma unlinktmp_left_mover :
  left_mover DeliverRestrictedAPI.step
             (DeliverOp.UnlinkTmp).
Proof.
  split; eauto.
  ...
Qed.
```

Figure 9: Example lemmas about movers that arise in verifying the implementation of the MailboxTmpAbs layer on top of the Deliver layer. `DeliverRestrictedAPI.step` refers to a restricted version of the semantics of the `Deliver` layer (using the protocol pattern from §6.2), where the filename of any file linked into a user's mailbox must contain the PID of the process that called `link()`.

into low-level file system operations from Figure 6, such as the pseudocode shown in Figure 1 and Figure 2 (except that our actual implementation is in Coq, which is not as easy to read as the Python-like pseudocode).

To produce runnable code, CSPEC extracts this code to Haskell using Coq's code extraction facility, and replaces the low-level operations with actual file system calls. An unproven driver, written in Haskell, interfaces with the network (e.g., accepts connections using sockets) and calls the appropriate top-level operations. To verify the driver would require verifying the parsing of SMTP and POP3 messages, which we didn't do because it has little to do with concurrency. Finally, the Haskell compiler produces an ELF executable.

5.4 Proving

Verifying the implementation entails proving that the code generated by `compile_op` correctly implements every high-level operation in terms of the lower-level operations. This includes proving that the code preserves the atomicity of high-level operations, given the atomicity of the lower-level operations. To make this task easier, CSPEC provides several proof patterns that encapsulate proof techniques to prove theorems about the behavior of a concurrent system.

For instance, the mover approach described in §4 is one such technique. It allows the developer to prove that certain operations are atomic. Figure 9 shows the lemmas needed to prove the atomicity of `deliver_core` of Figure 8. The lemmas state that `CreateWriteTmp` is a right mover and `UnlinkTmp` is a left mover, and the developer must write a proof in Coq (and checked by Coq) to show that this is true. CSPEC provides a general-purpose theorem (discussed in §6) that translates these developer-proven lemmas into a proof that the entire implementation of `deliver_core` executes atomically.

Note that `compile_op` in Figure 8 translates many operations one-to-one to lower-level operations. It is trivial to prove that they are atomic because the lower-level operations are atomic, and CSPEC does this automatically.

CSPEC chains the proofs of each layer's `compile_op` to provide an end-to-end proof that the resulting executable system meets the top-level atomicity specification.

6 CSPEC's proof patterns

CSPEC provides a library of proof patterns that help in proving that the code connecting two layers is correct. This section presents each proof pattern in turn.

6.1 Mover pattern

The key pattern provided by CSPEC for reasoning about concurrency is the *mover pattern*. As we saw in §4, this reduces the problem of reasoning about many interleavings (i.e., concurrent execution) to a combination of reasoning about just one interleaving (i.e., atomic sequential executions) and proving that certain operations are left- or right-movers.

For instance, consider the implementation of `Deliver` shown in Figure 8 as `deliver_core`. Running this implementation concurrently with other `Deliver` and `Pickup` operations can produce many interleavings, since there are no locks. It is not even possible to enumerate all the possible interleavings, since there can be an arbitrary number of concurrent processes.

Intuitively, we can reason about the execution of `deliver_core` by observing that the `link` operation is the commit point. That is, before `link` other processes are not affected (e.g., they cannot observe partially delivered messages), and after `link` other processes can observe the delivered message (if `link` succeeds).

Equivalence. To formalize this line of reasoning, CSPEC reasons about *equivalent executions*—that is, two interleavings that must produce the same trace of events. For example, changing the order of the `unlink` in `Deliver`, with respect to other processes, produces the same trace.

To prove the atomicity of a procedure using CSPEC's mover pattern the developer shows that certain operations are left- or right-movers with respect to other operations, and that the code of the procedure consists of these left- and right-movers operations in a certain order. Given these lemmas, CSPEC provides a theorem that proves the equivalence of other interleavings.

Movers. CSPEC models the concurrent execution of an overall system by repeatedly executing one operation from some process, which leads to a particular *execution sequence*. Treating an entire operation as a single transition captures the idea that operations are atomic. The choice of the process whose operation is executed at each point in this execution sequence determines a particular interleaving. By considering all processes at

```
Definition right_mover step `(opA : Op TA) :=
  forall `(opB : Op TB) st0 st1 st2 pidA rA pidB rB,
    pidA <> pidB ->
    step opA pidA st0 rA st1 ->
    step opB pidB st1 rB st2 ->
      exists st1',
        step opB pidB st0  rB st1' /\
        step opA pidA st1' rA st2.
```

Figure 10: Definition of the right mover.

```
Theorem trace_incl_movers : forall `(p : proc Op T),
  right_left_mover_pattern p -> trace_incl p (Atomic p).
```

Figure 11: Mover pattern theorem. `proc Op T` is a type denoting a procedure that returns a value of type `T` and can invoke operations described by `Op`. `Atomic p` denotes a procedure that atomically executes `p`.

each point in the execution sequence, CSPEC considers all possible interleavings between concurrent processes.

Figure 10 formally defines what it means for a certain operation, `opA`, to be a right-mover. Specifically, it considers every possible execution where `opA` is followed by some other operation, `opB`, from a different process (with process ID `pidB`). In this execution, `opA` changes the state from `st0` to `st1`, and `opB` changes the state from `st1` to `st2`. In order for `opA` to be a right-mover, it must be possible to swap `opA` with `opB` in this execution: that is, if `opB` ran first, it must produce some state `st1'` such that `opA` will then produce `st2`, and `opB` and `opA` produce the same return values `rB` and `rA` respectively. Left movers are defined similarly (there are some subtle differences that we discuss later). As an example, Figure 9 showed how one layer of CMAIL uses these definitions.

Showing that an operation O is a left- or right-mover requires considering how O interacts with every possible operation from another process. Layers help by making it possible to define the operations in a way that makes it easier to prove that other operations commute.

Composing movers. In order to reason sequentially about the execution of a procedure, its code must consist of a sequence of right-movers, followed by zero or one non-movers, followed by a sequence of left-movers. This structure allows CSPEC to show that any possible execution sequence is equivalent to one where the procedure executes sequentially, with no intervening operations from other processes. Specifically, CSPEC provides a theorem, shown in Figure 11, stating that any trace produced by procedure `p` is also produced by procedure `Atomic p` (which executes `p` in a single atomic step), as long as `p` follows the above mover pattern.

CSPEC proves this theorem by moving all of the right-movers to the right and all of the left-movers to the left, so that they appear to execute sequentially with the optional non-mover in the middle. The non-mover (for example, `link` in `Deliver`) is the commit point of the operation.

Left-mover challenges. Proving the theorem in Figure 11 is difficult, and required addressing several challenges with the formalization of left-movers.

First, operations can be left-movers just with specific arguments or just in specific states. For example, the implementation of `Pickup` first lists the files in a user's mailbox directory and then opens and reads the files one at a time, as shown in Figure 2. Here, the `open` operation is a left-mover only if called with the pathname of a file in the user's mailbox. (The implementation of `Pickup` holds a lock to prevent concurrent deletes, but does not prevent concurrent deliveries.) It is not a left-mover if called with a filename in the temporary directory, because opening a temporary file might succeed or fail depending on what a concurrent `Deliver` does in the temporary directory. Furthermore, `open` is a left-mover only if the file already exists *before* the other operation (i.e., the operation being re-ordered with respect to the `open`). Otherwise, this other operation might be `Deliver` creating the file in question in the mailbox.

CSPEC supports state- and argument-dependent left movers by restricting the states and arguments that have to be considered by the left-mover. Specifically, CSPEC requires the left-mover to consider only those states and arguments that can arise after executing the prefix of the operations leading up to the left mover. For instance, the procedure `p` shown in Figure 11 may be composed of several right-movers, followed by a non-mover, followed by several left-movers. The left-movers have to consider only the states that can arise after the right-movers and the non-mover have executed.

To take advantage of state-dependent left-movers, the developer first states an invariant that is established by executing the right-movers followed by the non-mover. The developer then proves a lemma that, starting from any state, executing the right-movers followed by the non-mover establishes this invariant. Finally, when reasoning about a left-mover, the developer can invoke this lemma to prove that the state observed by the left-mover satisfies the invariant.

Second, CSPEC's model of operations allows for operations to be *disabled*: that is, the semantics forbids an operation to execute in a given state. This is represented by a `step` relation that does not provide any legal transitions for a particular operation and a particular state. In the top-level and lowest-level layers such restrictions do not appear, because CMAIL can always deliver and pickup mail and the file system can always execute an operation (even if only to return an error). However, disabled operations are helpful in intermediate layers, in order to prove that other processes follow certain rules.

The simplest example is a lock that protects memory accesses. Reads and writes to memory protected by a lock commute with other threads, because those threads cannot access the locked memory. By taking advantage of the fact that reads and writes are disabled for other threads that do not hold the lock, CSPEC allows a proof that

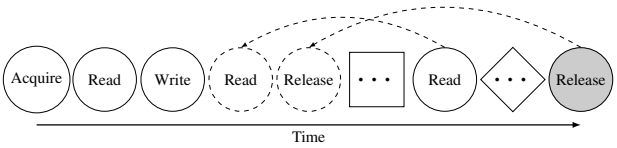


Figure 12: Use of left movers in an example process that acquires a lock, reads a variable, writes a variable, then reads a variable again and releases the lock. The gray shading of the `Release` on the right indicates that, although the `Release` is part of the code for the round process, the execution sequence shown is one where the `Release` never gets around to executing (e.g., due to other threads preempting it).

reads and writes are movers. (CSPEC's protocol pattern, described in §6.2, allows a developer to show that it is correct to assume that certain operations are disabled.)

Disabled operations complicate the notion of a left-mover, because moving an operation to the left requires showing that it can be executed earlier, which requires showing that it is enabled earlier. Consider a simple example shown in Figure 12. Reading from a locked memory region requires that the caller hold the lock. Moving the second read earlier requires showing that the caller holds the lock at that point. CSPEC deals with this by requiring a proof that a left-mover is *stably enabled*. This means that if the operation was enabled in a certain state (e.g., at the point where the second read actually ran in Figure 12), then it must be enabled in a prior state before another operation from a different process ran (e.g., in its dashed location in Figure 12). The read is stably enabled because the process must have held the lock, and no other process can acquire or release this process's lock.

The final challenge has to do with liveness. For example, the `Release` in Figure 12 is a left-mover, and we would like to use this fact to make the entire sequence of five operations into an atomic step. However, in our example, `Release` never actually ran (i.e., it is not part of the execution sequence). This might be because the scheduler is not fair and repeatedly ran other processes instead. How can we re-order the `Release` if it does not appear in the execution sequence to begin with?

To deal with this problem, CSPEC's proof considers all possible execution sequences. If an execution sequence contains the `Release`, the proof uses the fact that it is a left-mover to move it left. However, if an execution sequence does not contain the `Release` (i.e., if the `Release` never runs), then it is safe to insert that `Release` into the execution sequence. Stable enablement of left movers guarantees that `Release` is enabled at the point where we would like to insert it (i.e., the `Release` cannot have been waiting for another process to do something), and `Release` being a left-mover guarantees that other operations from this execution sequence will not be affected by inserting this `Release` (because they never saw the lock being released in the first place).

```
Definition op_abs :=
  forall '(op : Op T) st st' ST pid r,
    absR st ST ->
    lo_step op pid st r st' ->
      exists ST',
        absR st' ST' /\ hi_step op pid ST r ST'.
```

Figure 13: Definition of abstraction.

6.2 Protocol pattern

Proving that operations are left- or right-movers sometimes requires reasoning about what other processes will do, not just about what operations they have. In the lock example above, proving that memory accesses are movers while holding the lock requires knowing that other processes will not access the same memory while this process is holding the lock. To reason about such examples, CSPEC requires the developer to define a *protocol*, which is a restricted version of the `step` execution semantics that disables certain transitions. In the lock example, this restricted semantics requires that the caller hold the lock in order to read or write memory. With this restricted `step` relation memory accesses are movers, because other processes are not allowed to access the same memory location while not holding the lock.

In reality, nothing prevents another process from accessing memory without holding the lock. Thus, a proof that is sound to use the restricted semantics requires a proof that all users of the API correctly follow this protocol. Specifically, this entails proving that any execution of a process's code on the unrestricted semantics is also a valid execution on the restricted semantics.

In theory, this requires reasoning about many interleavings. In practice, however, the reason that a procedure follows a protocol is often simple (e.g., syntactically, the program never calls `Release` unless it called `Acquire` first). Thus, the proof needs only limited reasoning about the execution of other processes. In the locking example, proving that a process reads or writes memory only while holding a lock requires just one helper lemma: that other processes will not release a lock held by this process.

6.3 Abstraction pattern

To connect layers with different types of states, CSPEC provides an abstraction pattern. The abstraction pattern requires the developer to define an *abstraction relation* that connects low-level and high-level states, and to prove that every operation preserves this relation. This pattern is a specialized version of a standard simulation proof: it requires that the operations remain the same.

Figure 13 formally defines the proof obligation for the abstraction pattern. It requires a proof that, for every operation `op`, if `op` runs from state `st` to `st'` in the low-level semantics, and low-level state `st` corresponds to high-level state `ST` according to the abstraction relation `absR`, then there's a state `ST'` that corresponds to `st'` such

that the same op runs from ST to ST' with the same return value.

The rest of this subsection describes two stylized uses of the abstraction pattern that we have found particularly useful in developing CMAIL and the x86-TSO locked counter example.

Invariant. The abstraction pattern allows a developer to prove that a layer follows an invariant: some property of states at that layer that is maintained by that layer's semantics. This in turn can help the developer apply other patterns, such as movers or the protocol pattern.

Operationally, the developer first specifies an invariant by defining a layer whose semantics require the invariant to hold in the initial and final state of every operation; the operations and the type of state remain the same. The developer then defines an identity abstraction relation (connecting states one-to-one). Finally, the proof of the abstraction relation shows that, if the invariant holds in some state, running any operation results in a state that also satisfies the invariant.

Error state. The abstraction pattern can also allow a developer to defer reasoning about unreachable states by defining an explicit error state. This is useful at lower-level layers, which have insufficient information to prove that certain states are unreachable (e.g., because it is up to the implementation of higher layers to avoid those states). This is simpler than an alternative plan that fully describes what happens in these states, and allows subsequent layers to treat all of these error states identically.

Operationally, the developer defines a protocol that they expect to follow (much as in the protocol pattern from §6.2), and augments the state with a designated *error* state. The developer then modifies the execution semantics so that, if the protocol is not followed, the execution transitions into the error state. Once the execution enters the error state, it remains in that state forever.

To connect an error-state layer to a lower layer without an error state, the developer defines an abstraction relation that allows the high-level error state to correspond to any low-level state. To connect two layers with error states, the developer defines an abstraction relation that connects the error states at the two layers. To finally dismiss the error state, the developer uses the protocol pattern to show that an implementation never enters the error state, and thus the error state is unreachable.

6.4 Other patterns

Retry loop. CSPEC provides a specialized pattern for reasoning about retry loops. For example, when the mail server is delivering a message into a mailbox, it guesses a name that is unlikely to exist (using the current timestamp), and attempts to link the new message under that name. If `link` returns an error (i.e., the name already exists), CMAIL guesses a new filename and retries.

Component	Lines of code/proof
Core: processes, layers, etc.	4,594
Proof patterns	2,117
Helper: Maps, Sets, etc.	2,869
Total	9,580

Figure 14: Combined lines of code and proof for CSPEC components

The retry loop pattern requires a proof that the body of the loop either has the correct effect (such as delivering the message into a mailbox) and exits the loop, or has no effect and retries. This allows CSPEC to prove that executing the loop is equivalent to just running the body once, at exactly the right time (when it finally succeeds), because it can provably ignore all previous attempts (since they must have had no effect).

Partitioning. CSPEC provides a partitioning pattern to reason about disjoint parts of the state. For example, CMAIL has a separate mailbox for every user. Without explicit support for partitioning, the developer would need to reason about pairs of users at every layer of CMAIL—for instance, showing that an operation is a right-mover would require considering concurrent operations both on the same mailbox and on other mailboxes.

To use CSPEC's partitioning pattern, the developer implements and proves layer `A` on top of layer `B`, using CSPEC's other patterns, where `A` and `B` represent a single shard of the overall system state. For example, the core of CMAIL implements the per-user `MailServerPerUser` layer on top of the per-user `MailFSPath` layer, as shown in Figure 7. The developer must also specify how these shards are named (e.g., by string username in the case of CMAIL). The partitioning pattern turns this proven single-shard implementation into a proven implementation for many shards (e.g., all users in CMAIL).

As shown in Figure 7, cross-mailbox operations show up just at the top and bottom layers of the CMAIL stack. At the bottom layer, the proof must show that mailboxes are correctly partitioned in the file system—that is, each mailbox gets its own directory that is independent of all other mailbox directories. At the top level, the developer must specify and prove how the entire state of the system can be decomposed into per-user partitions. This is straightforward for CMAIL because the top-level abstract state (Figure 3) consists of a mailbox per user.

7 Implementation

We implemented CSPEC in Coq. Figure 14 shows the lines of code, specification, and proof for the major components. Developers implement, specify, and prove their concurrent software in Coq, and CSPEC produces executable code using Coq's extraction support to Haskell. Our prototype of CSPEC and CMAIL is available at https://github.com/mit-pdos/cspec.

One technical difficulty in implementing CSPEC is that Coq (like many other formal reasoning systems) makes it cumbersome to reason about infinite objects (i.e., Coq's `CoInductive`), as opposed to arbitrary-sized objects (i.e., Coq's `Inductive`). This made it hard for us to model the possibly infinite traces of events produced by the execution of a concurrent system.

To deal with this, we borrowed an idea from Lynch [29: §13], taking advantage of the fact that CSPEC is targeting only safety properties. A violation of safety can be observed in a finite prefix of the trace. Thus, we define trace inclusion in Coq for possibly infinite traces as trace inclusion for every finite prefix of that infinite trace.

8 Evaluation

This section answers five questions to evaluate CSPEC:

- Can CSPEC enable developers to specify, implement, and verify concurrent software? §8.1 answers this in the context of CMAIL, and §8.2 demonstrates that CSPEC's patterns are also applicable for a different style of concurrency: namely, weak shared memory.
- Can software developed using CSPEC actually achieve speed-ups by taking advantage of concurrency? (§8.3)
- How much effort is required to use CSPEC? (§8.4)
- How important are CSPEC's patterns? (§8.5)
- What are the trusted components of CSPEC and CMAIL? (§8.6)

We answer the above questions by exploring two case studies built using CSPEC: a concurrent mail server (CMAIL) and a concurrent counter that uses locks implemented on top of an x86-TSO memory model.

CMAIL is a simple but complete mail server that supports SMTP and POP3. It runs on top of any file system on Linux and we have tested its compatibility with several SMTP and POP3 clients, including the SMTP library in Go, and the postal and rabid mail server benchmarks. CMAIL lacks sophisticated features found in standard mail servers, such as spam filtering, logging, TLS support, etc.

8.1 Verifying CMAIL

To show that CSPEC enables reasoning about concurrency, we give examples of concurrency from our two case studies. This subsection describes the examples of concurrency from CMAIL, and the next subsection describes our experience verifying an atomic counter on top of x86-TSO weak memory.

Figure 15 summarizes the examples of concurrency from CMAIL, by describing pairs of processes that might run concurrently, the state that they might access concurrently, the plan for dealing with this concurrent execution, and how we as developers were able to use CSPEC to formally reason about the correctness of this concurrent interaction. The rest of this subsection describes these examples in more detail.

Deliver/Deliver: temp directory. Accepting an incoming message requires CMAIL to first write it to a temporary directory. However, there can be concurrent deliveries writing to the same directory at the same time. For correctness, a CMAIL process includes its PID in the names of its temporary files, which ensures two processes never conflict on files in the temporary directory. In CSPEC, we formally reason about this by showing that operations on the temporary directory always commute between different processes, because they have different PIDs in the filenames.

Pickup/Delete. If a user has two connections to CMAIL, and deletes a message on one connection while picking up messages via another connection, then the code for pickup, which lists and picks up messages, might discover halfway through that it cannot read a message file because the file has been deleted. CMAIL deals with this by acquiring a lock (using POSIX `flock`) on the user's mailbox, in both pickup and delete (but not in deliver; concurrency between deliver and pickup will be discussed next). We reason formally about this in CSPEC by first proving that CMAIL follows a protocol that requires holding a lock to delete any messages, and then showing that reading an existing message file is a both-mover while the lock is held.

Deliver/Pickup by another user. When CMAIL delivers or picks up mail for different users in different processes the concurrency plan is easy: these operations are independent because they operate on different mailboxes. In CSPEC, we show that operations on different mailboxes are commutative.

Deliver/Pickup by same user. A user can pick up (list and read) the messages in their mailbox while CMAIL is concurrently delivering new messages to that same mailbox (by creating files). CMAIL handles this like Maildir: it first creates new messages in a temporary directory, and then atomically renames them into the mailbox directory. When a user picks up their mail, CMAIL first calls `readdir` to list the files in the mailbox, and then reads the files in a loop. This is correct even in the presence of concurrent deliveries, because deliveries never delete existing files. To reason formally about this in CSPEC, we show that creating temporary files during delivery is a right-mover, and the atomic rename by delivery is a non-mover. On the pickup side, `readdir` is a non-mover, but all subsequent reads of existing files are left-movers.

Deliver/Deliver: files in mailbox, plan 1. Concurrent deliveries into the same user mailbox must ensure they pick different file names for the new messages. CMAIL implements two plans for this scenario, to demonstrate

Process 1	Process 2	State	Concurrency plan	CSPEC approach
Deliver message	Deliver message	Temp. directory	File names based on PID	Both-movers due to commutativity
List mailbox	Delete a message	Files in mailbox	Lock the mailbox directory	Protocol: both-movers while holding lock
Deliver to one user	Pickup by another user	Files in mailbox	Per-user directories	Both-movers due to commutativity
Deliver to one user	Pickup by same user	Files in mailbox	Atomic rename / readdir	Non-mover rename, non-mover readdir
Deliver to one user	Deliver to same user	Files in mailbox	List files by PID and pick next	List-per-PID is a mover
Deliver to one user	Deliver to same user	Files in mailbox	Retry link to random filename	Retry loop

Figure 15: Examples of concurrency from CMAIL supported by CSPEC.

how different approaches can work. In the first approach (which differs from Figure 1), filenames in the mailbox directory are based on the PID of the process that delivered the message. To pick an available filename, the delivery process calls `readdir` to list the directory, and chooses the next available filename that contains its PID.

Formally reasoning about this turns out to be tricky in two ways. First, `readdir` is not a mover, because its results can be affected by concurrent deliveries. To use mover-based reasoning, we implemented a function that filters the output of `readdir` and returns only the filenames of the caller's PID. This PID-filtered `readdir` function is a both-mover, because concurrent deliveries by different processes have filenames with different PIDs.

Second, in the presence of concurrent message deletion, even PID-filtered `readdir` is not quite a mover. We solve this by allowing it to return a superset of files that exist: that is, it must return all files that exist but can also return some non-existent files. This suffices because a filename that is not in the superset is guaranteed to not exist. This PID-filtered `readdir` is a right-mover in the presence of deletion (though not a left-mover), and so we can use the mover pattern to reason about its concurrent execution.

Deliver/Deliver: files in mailbox, plan 2. The second plan we implemented for concurrent deliveries to the same mailbox is to pick a random filename and try using it. POSIX `link` returns an error if the file already exists, so in case of an error CMAIL picks a new random filename (actually, it uses the current timestamp) and retries. To reason about this we use the retry pattern, showing that `link` either succeeds or returns an error and has no effect.

8.2 Verifying a counter on x86-TSO

CMAIL's concurrency model is based on processes with private memory sharing a file system. To demonstrate that CSPEC can also be used to reason about other concurrency models, we developed a model of x86-TSO [34], the predominant memory model of x86 processors. On top of x86-TSO, we implemented a lock, and used the lock to implement a counter. The lock implementation is a loop around an atomic test-and-set instruction, which includes an implicit write barrier (on x86, this corresponds to a `LOCK` prefix on the test-and-set instruction). We used 10 layers to verify this counter, as shown in Figure 16.

The top layer is a counter with two atomic operations: increment and decrement. The bottom layer, `TSO`, models x86-TSO: there is a shared memory and a per-core store buffer, and individual cores can issue reads, writes, or atomic test-and-set instructions, as well as perform a write barrier to flush that core's store buffer. Every operation at this bottom layer allows any core to flush any part of its store buffer at any time.

One challenge in the `TSO` layer is that background flushes of store buffers can happen on any core at any time. To help address this challenge, we showed that the `TSO` layer is equivalent to the `TSODelayNondet` layer which does not allow store buffer flushes on write (instead, postponing them to a subsequent read, barrier, or test-and-set).

The `LockOwner` layer introduces abstract state to keep track of which core owns the lock, using the abstraction pattern. Our intention is that the lock protects reads and writes to a shared memory location. However, this proper use of the lock is not established until a higher layer (namely, `Lock`). As a result, the `LockOwner` layer uses an explicit error state (§6.3) to indicate when the locking rules are not being followed. This error state is proven to be unreachable in the `Lock` layer (using the protocol pattern).

The `LockInvariant` layer additionally tracks the previous lock owner as part of the state. This is necessary because the implementation of lock release does not issue a write barrier. As a result, even though the lock may have been released, the lock value in shared memory may still appear to be locked, and pending writes to shared data are also in some core's store buffer. By tracking the previous lock owner, the `LockInvariant` layer states an invariant that either there are no pending writes to the lock or shared data in any core's store buffer, or they are in the previous lock owner's store buffer. The next layer, `SeqMem`, builds on this invariant to present a sequentially consistent view of shared memory, abstracting away the store buffer details.

The `RawLock` layer introduces an `Acquire` operation that waits until it can acquire the lock. This layer is implemented on top of `SeqMem` by repeatedly trying to acquire the lock in a loop. The proof is constructed with the help of CSPEC's loop pattern.

Layer name	Operations	State	Semantics	Pattern
Counter	Inc, Dec	Counter value	Atomic Inc and Dec	Abs
LockedCounter	Inc, Dec	Counter value + lock	Atomic Inc and Dec	Mov
Lock	Read, Write, Acquire, Release	SC memory + lock	Read/Write allowed only while holding lock	Prot
RawLock	Read, Write, Acquire, Release	SC memory + lock	Read/Write allowed any time	Loop
SeqMem	Read, Write, TryAcquire, Clear	SC memory + lock	Single value in memory, no SBs	Abs
LockInvariant	Read, Write, TryAcquire, Clear	Mem + SBs + cur/prev LOs	SBs empty except current or prev lock owner	Abs
LockOwner	Read, Write, TryAcquire, Clear	Mem + SBs + current LO	TSO + error state for violating lock protocol	Abs
TAS_TSO	Read, Write, TryAcquire, Clear	Mem + SBs	TryAcquire grabs lock; Clear releases lock	Mov
TSODelayNondet	Read, Write, TestAndSet, Barrier	Mem + SBs	Reduced number of background SB flushes	Abs
TSO	Read, Write, TestAndSet, Barrier	Mem + SBs	SB may choose to flush on every operation	

Figure 16: Layers used for verifying the x86-TSO locked counter. The operations column describes the `Op` type for that layer. The state column describes the abstract state, `State`, over which the layer's semantics are defined. The semantics column describes the semantics. The pattern column indicates which CSPEC proof pattern is used in connecting adjacent layers. "SC" stands for sequentially consistent. "SB" stands for store buffer. "LO" stands for lock owner.

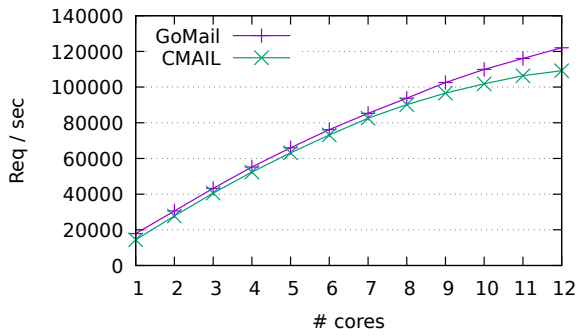


Figure 17: Throughput of CMAIL with a varying number of cores.

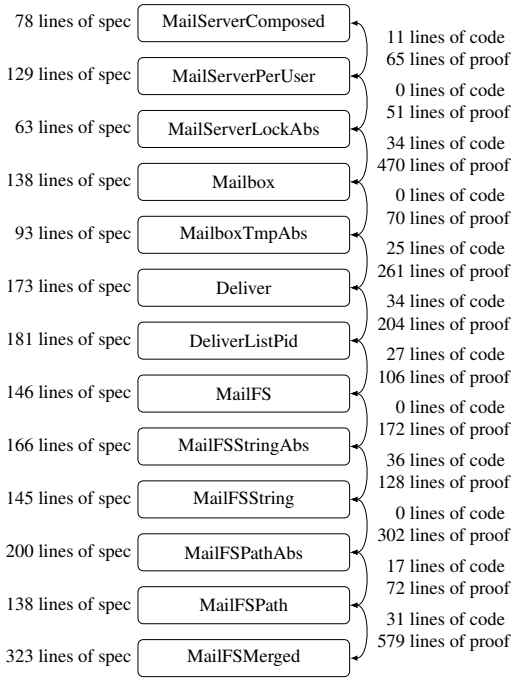


Figure 18: Combined lines of code and proof for CMAIL layers. The number next to arrow indicates number of lines of code and proof for the implementation connecting two layers.

8.3 Speedup

To demonstrate that CMAIL can take advantage of multiple cores because it executes concurrently, we run a mixed workload of SMTP deliveries of new messages and POP3 requests that read and delete messages. The mix is an equal ratio of new messages being delivered and existing messages being read and deleted. Each request (delivery or pickup) chooses one of 100 users at random. Although CMAIL supports full-fledged SMTP and POP3 over the network, we simulated SMTP and POP3 requests on the same machine to stress CMAIL's scalability. We ran the experiment on a server with two Intel Xeon CPU, each with 6 cores running at 3.47 GHz. To keep the disk from being the bottleneck, we ran CMAIL on Linux tmpfs. To compare the performance of CMAIL to that of an unverified implementation, we implemented an equivalent mail server in Go, called GoMail, and measured its performance in the same setup.

Figure 17 shows the performance (in requests per second) for different numbers of cores of both CMAIL and GoMail. The results show that CMAIL scales well with more cores. This is because tmpfs can execute the file system calls of the different CMAIL processes in parallel. In terms of absolute performance, CMAIL achieves 81-97% of GoMail's throughput, depending on the number of cores.

8.4 Effort

Figure 18 shows the size of CMAIL: the lines of Coq code to specify each layer (i.e., define operations, state, and semantics) and the lines of Coq code required to connect layers (i.e., implement one layer in terms of a lower layer and prove the correctness of that code). Developing CSPEC and CMAIL took two people ∼6 months of part-time effort.

The figure shows that the effort required per layer is modest. Each layer spec is 100-200 lines of Coq code, which are largely repetitive, with only small differences between adjacent layers. Informally, the specs of adjacent layers differ in roughly half the lines, and even the differing lines are often similar (e.g., an extra state component is added everywhere). Better language support, perhaps

Proof pattern	# of uses in CMAIL	# of uses in x86-TSO
Movers	6	2
Abstraction	5	5
Protocol	3	1
Partitioning	2	0
Retry loop	1	1

Figure 19: Use of proof patterns in CMAIL and the x86-TSO example.

along the lines of CIVL's [22], could eliminate the repetition. A layer often maps a high-level operation directly onto a low-level operation, so it should be sufficient to write the spec only once. For example, CMAIL's `GetPID` is the same in each of the 13 layers.

The code and proof is sometimes shorter than the layer spec because some code takes advantage of CSPEC's patterns so well that it requires little additional proof effort. This is particularly true for the abstraction pattern that introduces additional state not seen at a higher layer (e.g., adding state for a lock that is hidden at a higher layer).

The most significant code and proof effort connects the MailServerLockAbs and Mailbox layers, where CMAIL implements atomic pickup. This requires a proof that pickup's file reads are left-movers, and inductive reasoning about a loop that reads all files. This is particularly hard because the file read is a state- and argument-dependent left mover, which requires reasoning about the set of files that exist in the system after `readdir` returns.

Evolution. To evaluate how hard it is to make incremental changes to a verified system in CSPEC, we report the effort it took us to make several significant changes to CMAIL as we were developing it. Initially our mail server supported POP3 retrieval but not deletion. Adding deletion support took about a day: we had to change some mover proofs because deletion made certain operations into non-movers. Our initial mail server used plan 1 to choose unique file names in a mailbox (see §8.1); implementing plan 2 using retry loops with `link` took us about a day. Finally, adding support for multiple users took us about a week. After a day, we realized that manually adding users to each layer was too tedious, and spent a week developing the partitioning pattern in CSPEC. Afterwards, supporting multiple users took about a day.

8.5 Patterns

Figure 19 shows the number of uses of a proof pattern in CMAIL and in the x86-TSO example. Typically each layer uses one proof pattern, but a few layers are split into several modules, each module using a distinct proof pattern. The results show that all patterns are important; that movers is the most commonly used pattern in CMAIL; and that abstraction is the most common pattern in x86-TSO.

8.6 Trusted computing base

Whether CMAIL does the right thing depends on several unverified assumptions and components. The first assumption is that the top-level specification (Figure 3) captures the right behavior. Second, the specification of the bottom layer (Figure 6) must be an accurate model of the underlying file system. Finally, the Haskell runtime and interpreter used to run CMAIL must behave appropriately.

CMAIL also requires Coq to be sound, but inside of Coq, CMAIL and CSPEC are fully proven. We used the `Print Assumptions` command in Coq to verify that the end-to-end theorem about correctness of CMAIL does not depend on any unproven axioms (aside from standard assumptions like Coq's functional extensionality).

9 Conclusion

CSPEC is a framework for verifying concurrent systems software. It uses mover types to simplify reasoning about both lock-based and lock-free concurrency, with the first fully machine-checked proofs. To further simplify proofs, CSPEC has layers and a library of proof patterns. CMAIL demonstrates that CSPEC can verify all the concurrency patterns in a Maildir-like mail server. Furthermore, we demonstrate that CSPEC's proof pattern can also be used to prove an atomic lock-based counter on top of x86-TSO shared memory. CMAIL achieves speedup on a multicore machine due to concurrency. We hope that CSPEC and its ideas will help others to verify concurrent software.

Acknowledgments

Thanks to Adam Chlipala, the PDOS research group, the anonymous reviewers, and our shepherd, Jon Howell, for improving this paper. This research was supported by NSF awards CNS-1563763 and CCF-1836712, and by Google.

References

[1] R. Affeldt and N. Kobayashi. Formalization and verification of a mail server in Coq. In *Proceedings of the 1st International Symposium on Software Security (ISSS)*, pages 217–233, Tokyo, Japan, Nov. 2002.

[2] R. Affeldt, N. Kobayashi, and A. Yonezawa. Verification of concurrent programs using the Coq proof assistant: A case study. *IPSJ Digital Courier*, 1: 117–127, Jan. 2005.

[3] D. Bernstein. Using maildir format, 2003. http://cr.yp.to/proto/maildir.html.

[4] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran. Reasoning about TSO programs using reduction and abstraction. arXiv:1804.05196

[cs.LO], Apr. 2018. Available at https://arxiv.org/abs/1804.05196.

[5] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

[6] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, Aug. 2009.

[8] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-2019, Microsoft Research, Feb. 2009.

[9] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, Edinburgh, UK, July 2010.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, Nov. 1978.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009.

[12] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009.

[13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, Nov. 2016.

[14] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[15] C. Hawblitzel. Personal communication, email, Mar. 2018.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, Oct. 2015.

[17] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, San Francisco, CA, July 2015.

[18] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, Feb. 2015.

[19] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60 (7):83–92, July 2017.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, Oct. 1983.

[21] J. Kim, V. Sjöberg, R. Gu, and Z. Shao. Safety and liveness of MCS lock–layer by layer. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 273–297, Nov. 2017. http://flint.cs.yale.edu/flint/publications/mcslock-tr.pdf.

[22] B. Kragl and S. Qadeer. Layered concurrent programs. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, Oxford, UK, July 2018.

[23] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] B. Lampson. Principles of computer systems, 2006. http://bwlampson.site/48-POCScourse/48-POCS2006Abstract.html.

[25] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[26] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 357–370, St. Petersburg, FL, Jan. 2016.

[27] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), Dec. 1975.

[28] J. Lorch et al., 2016. https://github.com/Microsoft/Ironclad/tree/concur-tree/ironfleet/src/Dafny/Distributed/Common/Reduction.

[29] N. Lynch. *Distributed Algorithms*. Elsevier, 1996.

[30] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[31] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, Cambridge, MA, Nov. 1988.

[32] G. Ntzik. *Reasoning About POSIX File Systems*. PhD thesis, Imperial College London, 2017.

[33] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. In *Proceedings of the 45th ACM Symposium on Principles of Programming Languages (POPL)*, Los Angeles, CA, Jan. 2018.

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[35] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[36] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*, Apr. 2018. URL https://doi.org/10.5281/zenodo.1219885.

[37] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

Proving confidentiality in a file system using DISKSEC

Atalay İleri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

SFSCQ is the first file system with a machine-checked proof of security. To develop, specify, and prove SFSCQ, this paper introduces DISKSEC, a novel approach for reasoning about confidentiality of storage systems, such as a file system. DISKSEC addresses the challenge of specifying confidentiality using the notion of *data noninterference* to find a middle ground between strong and precise information-flow-control guarantees and the weaker but more practical discretionary access control. DISKSEC factors out reasoning about confidentiality from other properties (such as functional correctness) using a notion of *sealed blocks*. Sealed blocks enforce that the file system treats confidential file blocks as opaque in the bulk of the code, greatly reducing the effort of proving data noninterference. An evaluation of SFSCQ shows that its theorems preclude security bugs that have been found in real file systems, that DISKSEC imposes little performance overhead, and that SFSCQ's incremental development effort, on top of DISKSEC and DFSCQ, on which it is based, is moderate.

1 Introduction

Many security problems today stem from bugs in software. Although there has been significant effort in reducing bugs through better testing, fuzzing, model checking, and so on, subtle bugs remain and continue to be exploited. Machine-checked verification is a powerful approach that can eliminate a large class of bugs by proving that an implementation meets a precise specification.

Prominent examples of machine-checked security proofs include verification of strict isolation (with confidentiality) for an OS kernel in CertiKOS [15], seL4 [26], and Komodo [17], as well as security proofs in Ironclad [20] about applications like a password hasher and a notary service. However, proving the security of systems with rich sharing semantics, such as file systems, is an open problem. For example, unlike prior examples that focus on strict isolation without controlled sharing, users in a file system can share files with one another, and the underlying implementation has shared data structures (such as a buffer cache or write-ahead log) that contain data from different users.

Proving security for a file system requires addressing two key challenges. The first challenge lies in *specifying* security. Integrity can be expressed as simply as a functional correctness property. Confidentiality is more challenging to specify. For example, consider a natural specification for `readdir`, which allows the file system to return the names in a directory in any order. This nondeterminism could be abused by a buggy or malicious file system to leak confidential file data through careful manipulation of the order of `readdir` results. Furthermore, nondeterminism is essential to a file system, because file systems must deal with crashes, which can occur nondeterministically at any time.

One approach to specifying confidentiality is to formulate it as a noninterference property, such as in most information-flow-control systems. This means that the execution of one process (a potential victim processing confidential data) cannot influence the execution of another process (an adversary trying to learn that data). Noninterference can be stated concisely, and is easy for applications to use. However, information-flow-control style guarantees are stronger than what file systems aim for. Instead, file systems aim for weaker notions of confidentiality, along the lines of discretionary access control on files that reveal some metadata, such as file lengths.

A second challenge lies in proving confidentiality. Confidentiality is a "two-safety" property [34], which requires reasoning about *pairs* of executions to show that an adversary cannot observe any differences correlated with confidential data. However, reasoning about pairs of executions is more complicated than reasoning about a single execution, which is sufficient for proving integrity and functional correctness.

This paper presents DISKSEC, an approach for proving the security, and specifically confidentiality, for storage systems, such as file systems. The paper demonstrates the benefits of DISKSEC by developing, specifying, and proving the security of a file system in a prototype called SFSCQ, based on the DFSCQ file system [13].

DISKSEC addresses the specification challenge by using a notion of *data noninterference* that both matches what file systems aim to provide and is concise and easy to use for applications. Data noninterference requires that an adversary's execution be independent of the contents of individual files, but it allows the adversary to observe other metadata, such as file length and directory entries, and allows for discretionary access control (i.e., a user can choose to disclose their data).

To address the challenge of proving security, DISKSEC factors out reasoning about confidentiality from all other properties, such as functional correctness. DISKSEC does so by introducing a notion of sealed blocks. This builds

on the intuition that file systems do not look inside of the blocks that represent user file contents. As a result, DISKSEC is able to treat confidential file blocks as opaque in much of the file-system code, greatly reducing the need for manual proofs of two-safety that consider pairs of executions. The only manual proofs of two-safety are in the top-level `read` and `write` system calls.

We implemented DISKSEC and SFSCQ in the Coq proof assistant [35]. All proofs of security are machine-checked by Coq, eliminating the possibility of bugs that violate SFSCQ's specification. An evaluation of SFSCQ shows that its specifications are complete enough to prove confidentiality of a simple application. The evaluation also shows that DISKSEC's approach allowed us to develop SFSCQ with a modest amount of effort, and that SFSCQ achieves comparable performance to the DFSCQ file system that it is based on.

The contributions of this paper are:

- SFSCQ, the first file system with a machine-checked proof of confidentiality. SFSCQ has a concise specification that captures discretionary access control using data noninterference, and deals with nondeterminism due to crashes.
- DISKSEC, an approach for specifying and proving confidentiality for storage systems that reduces proof effort. DISKSEC uses the idea of sealed blocks to factor out reasoning about confidentiality from most of the file system code.
- An evaluation that demonstrates that DISKSEC's approach leads to negligible performance overheads in SFSCQ, that it precludes the possibility of confidentiality bugs that have been found in existing file systems, and that SFSCQ's specification allows applications to reason about their confidentiality.

Our SFSCQ prototype has several limitations. Since it relies on Coq's extraction to Haskell, inherited from DFSCQ, its trusted computing base (TCB) includes the Haskell runtime and compiler. The version of SFSCQ with fully machine-checked proofs does not support changing permissions. A newer version of SFSCQ supports dynamic permissions but has a few proofs that have not been repaired to reflect this change. Finally, SFSCQ's access-control mechanisms are relatively simple, supporting owned and public files but not groups or separate read and write permissions.

2 Related Work

DISKSEC builds on a large body of prior work in several dimensions, as we discuss in the rest of this section.

Data noninterference. DISKSEC's notion of data noninterference builds on prior work on formalizing noninterference properties [19, 25, 26, 29, 30, 32]. Specifically, data noninterference can be thought of as a specialization of abstract noninterference [18], relaxed noninterference [24], or observation functions [15]. One difference in our approach is that data noninterference stops at the file-system API boundary; applications are not subject to data-noninterference policies. This matches well the traditional discretionary access-control policies enforced by file systems.

Formalizing data noninterference requires reasoning about two executions, since confidentiality is a two-safety property [34]. In this context, our contribution lies in a specification and a proof style based on sealed blocks that helps us prove a data-noninterference two-safety property about the file system.

Machine-checked security in systems. Several prior projects have proven security (and, specifically, confidentiality) properties about their system implementations: seL4 [23, 26], CertiKOS [15], and Ironclad [20]. For seL4 and CertiKOS, the theorems prove complete isolation: CertiKOS requires disabling IPC to prove its security theorems, and seL4's security theorem requires disjoint sets of capabilities. In the context of a file system, complete isolation is not possible: one of the main goals of a file system is to enable sharing. Furthermore, CertiKOS is limited to proving security with deterministic specifications. Nondeterminism is important in a file system to handle crashes and to abstract away implementation details in specifications.

Ironclad proves that several applications, such as a notary service and a password-hashing application, do not disclose their own secrets (e.g., a private key), formulated as noninterference. Also using noninterference, Komodo [17] reasons about confidential data in an enclave and showing that an adversary cannot learn the confidential data. Ironclad and Komodo's approach cannot specify or prove a file system: both systems have no notion of a calling principal or support for multiple users and there is no possibility of returning confidential data to some principals (but not others). Finally, there is no support for nondeterministic crashes.

Information flow and type systems. Another approach to ensuring security is to rely on types or runtime enforcement mechanisms. Although this does not give a machine-checked theorem of security, we build on aspects of this approach, namely, the sealed disk has typed blocks.

Type systems and static-analysis algorithms, as with Jif's labels [27, 28] or the UrFlow analysis [14], have been developed to reason about information-flow properties of application code. However, these analyzers are static and would be hard to use for reasoning about data structures inside of a file system (such as a write-ahead log or a buffer cache) that contain data from different users.

Bug description	Filesystem(s)	year
anyone can change POSIX ACLs	btrfs [5], gfs2 [3]	2010
anyone can change POSIX ACLs	NFS [8]	2016
file permissions can be changed by writing to hidden file	reiserfs [2]	2010
truncated data can be accessed	btrfs [7]	2015
crash can expose deleted data	ext4 [9]	2017
crash can expose data	ext4 [22]	2014
can overwrite append-only file	ext4 [4], btrfs [6]	2010
can overwrite arbitrary files	ext4 [1]	2009

Figure 1: Bugs in various Linux file systems that can lead to data-disclosure or integrity violations.

Dynamic tools, such as Jeeves and Jacqueline [37, 38] and Resin [39], deal with dynamic data structures but require sophisticated and expensive runtime enforcement mechanisms. DISKSEC avoids the overhead of runtime enforcement and an additional trusted runtime checker.

Formalizing file-system security. Prior work has extensively studied the security guarantees provided by file systems, both formally and informally [10]. However, none of the prior work articulated a precise, machine-checkable model and specification for file-system security.

Symbolic models of cryptography. Our proof strategy is related to techniques introduced to reason about cryptographic protocols. Many cryptographic-protocol proofs are done in the Dolev-Yao model of perfect cryptography [16]. There programs are modeled as algebraic expressions, which developers reason about using equational axioms, like that decryption is the inverse of encryption, when called with identical symmetric keys. No equations allow breaking encryption without knowing the key. This model is attractive for its simplicity, and protocol-analysis tools like ProVerif [11] and Tamarin [33] build on it. HACL* [40] uses a similar proof strategy for proving its cryptographic library. DISKSEC's block-sealing abstraction extends this idea with the notion of a permission associated with each sealed block.

3 Motivation: bugs

File systems are an important building block for applications, which rely on the file system for security. For example, a mail server relies on the file system to ensure that data written to one user's mailbox file does not end up in some other user's mailbox file. Unfortunately, file systems have had bugs that allowed for data disclosure or modifying other users' files: we list several such bugs in Figure 1. In this section we describe several of these bugs in more detail.

File-system data leak. ext4 has an optimization called delayed allocation where new blocks for files are not actually allocated (but simply tracked) until they must be flushed to disk. It is important that even after a crash, blocks allocated in this manner have their new data written before ending up as part of the file; otherwise the old data in the block is leaked, potentially disclosing data from any user. For some time ext4 used its write-ahead log to ensure the new data was written atomically with the metadata changes to the file. An optimization introduced in 2012 removed this write-ahead logging [36], reasoning that the new data was always written to disk immediately with delayed allocation, before flushing the log. This optimization is incorrect: the disk may reorder writes so that the journal is actually written to disk first, exposing the old data on crash; the bug was fixed in 2016 by restoring the old behavior of writing the newly allocated blocks through the write-ahead log.

Access-control checks. File systems implement sophisticated policies for controlled sharing, such as file permissions, append-only or read-only files, and shared directories. It is easy for file-system developers to make mistakes in implementing these policies. For example, several file systems forgot to correctly implement append-only files when the file was being modified through a special interface for efficiently moving file data [4, 6]. In these examples, the file system did not read or write the file data itself but instead changed the data-block pointers inside of the file's inode. Another example is the privileged `nfsd` daemon, which forgot to check permissions when local users changed POSIX ACLs on a file [8]. A final example is a file system that stored metadata (including ACLs) in a separate file but failed to prevent users from directly modifying that separate file [2].

4 Goal

The goal of DISKSEC is to use machine-checked verification to ensure the absence of security bugs in file systems. Using a proof assistant (Coq) to check our proofs ensures that we consider all possible corner cases in our implementation when proving that it meets our specification. Thus, as long as our specification excludes the possibility of certain bugs, such as the ones described in the previous section, Coq will provide a high degree of assurance that no such bugs can exist in the implementation.

4.1 Threat model

From the perspective of verification, we would like to have confidence that the file system is secure purely based on the file system's security specification. This means that we have to treat the developer of the file system with an adversarial mindset. This subsumes all possible bugs that a well-meaning but error-prone developer might introduce into the implementation.

As a result, our threat model is that the adversary both develops the file system and runs an adversarial application on top of the file system in an attempt to obtain

confidential file data. However, the adversary does provide a proof that their file-system implementation meets our security specification. The potential victim runs on top of the same file system but sets their permissions so that the confidential files are not accessible to the adversary's process. Our goal is to ensure that the security specification is so strong that it prevents leaks even when the file-system developer is colluding with adversarial processes running on top of the file system.

Our threat model is focused on proving that the file-system implementation has no confidentiality bugs, rather than proving the absence of bugs in the environment outside of the file system. Thus, we assume that our model of how the file-system implementation executes is correct. That is, we are not concerned with bugs in unverified software or hardware outside of the file system, or users mounting malicious disk images. We do prove that `mkfs` produces a correct image, but ensuring confidentiality on top of an intentionally corrupted file system image is difficult, even without formal verification. We also do not reason about timing channels, as we do not model time.

4.2 Challenges

The most difficult aspect of formally proving the security of a file system lies in guaranteeing confidentiality. This is difficult for several reasons.

Two-safety. First, proving confidentiality is more difficult than proving functional correctness: as mentioned earlier, confidentiality is a two-safety property. Functional correctness is a one-safety property because a violation of functional correctness can be demonstrated by a single execution. For instance, if an application wrote one byte to a file and then read back a different byte, this single execution shows that the file system is incorrect. Thus, functional correctness of a file system is a theorem that says that all executions meet the spec (i.e., there are no such violations). Integrity properties, such as ensuring that one user cannot corrupt another user's data, are an example of a one-safety functional-correctness property and can be handled using standard verification techniques.

In contrast, demonstrating a violation of confidentiality requires two executions, where the results observed by an adversary differ depending on the secret data. For instance, consider a file system with block-level deduplication that also exposes the number of free blocks. An adversary who wants to learn the contents of a victim's file could write their guess for the victim's block into the adversary's own file and then check whether the number of free blocks stayed the same or decreased. If the file system implemented deduplication across users, this attack allows an adversary to learn whether their guess block was already present in the file system, thus inferring whether the victim has that data.

In the above example, looking at a single execution does not allow one to directly conclude that data was leaked, because the system appears to be functioning correctly. Determining that data is leaking requires one to consider a pair of executions, in which the adversary performs the same operations, but the confidential user data is different. If these two executions produce different adversary-observable results, the adversary is able to infer information about confidential data.

By stating confidentiality as a two-safety property, the above deduplication example would violate confidentiality, and thus could not appear in an implementation that was proven to achieve confidentiality. Specifically, suppose the starting states of the two executions differed in the contents of a confidential file, where in one execution the file matched the adversary's guess and in the other execution it didn't match. In this case, the number of free blocks returned by the adversary would differ in the two executions, which would not be allowed by the confidentiality definition.

Nondeterminism and probabilities. Another complication in proving confidentiality lies in the fact that many specifications, including those in the file system, are nondeterministic. Some nondeterminism is unavoidable because file systems must deal with crashes (e.g., due to power failure), which can occur at any time. Thus, it is impossible to know what are the exact contents of the disk after a crash; the on-disk state could reflect any prefix of the writes issued by the file system. Modern disks complicate this situation even further by buffering writes in memory inside the disk controller; as a result, the writes can be made durable out-of-order, and the state of the disk after a crash might reflect some out-of-order writes. Even in the absence of crashes, the file system implementation may want to use randomness (e.g., to randomize directory hash tables), which makes the execution nondeterministic.

Other nondeterminism comes from specifications that hide irrelevant details. For instance, the inode allocator in the file system does not specify which precise inode number will be returned; instead, its specification simply states that it will return *some* inode number that is not already in use. As another example, the specification for `readdir` in a file system likely allows the files in a directory to be returned in any order. The use of nondeterminism is important for keeping specifications concise and for allowing implementations to change (e.g., to implement performance optimizations) without modifying the specification.

Any nondeterminism is a potential leak of confidential data. The nondeterministic specification of the block allocator from above does not preclude the allocator from leaking confidential data, because it could, in theory, choose the next inode number based on the confiden-

tial contents of files, without violating its specification (i.e., still returning some unused inode number). Similarly, the nondeterministic specification for `readdir` is also not a good confidentiality specification, because a bug might cause the order of entries returned by `readdir` to be affected by the contents of some confidential file.

Even the nondeterminism associated with the state of the disk after a crash can be taken advantage of by an adversarial file-system implementation to leak data. For instance, a high-performance file-system specification allows the file system to delay flushing data to disk. An adversarial implementation could choose whether to flush data immediately or defer the flush based on one bit of confidential data from a victim's file. To take advantage of this, an adversary could wait for the system to crash and, after the crash, check whether any writes appear to have been lost. If so, the adversary concludes the file system must have deferred the writes, which would have only happened if the confidential bit was zero. This, in turn, can allow the adversary to infer confidential bits.

More generally, the possibility of leaking confidential data arises because nondeterministic specifications capture what *might* be possible, but an adversary may have more precise information about the actual *probabilities* of different outcomes. For instance, consider a hypothetical system call that returned random data. An adversarial file-system implementation could leak confidential data through this system call by sometimes returning uniformly random data and sometimes returning confidential data from some file. A naïve view of this system call might be that, since any return value is possible, this system call is not leaking any data. However, an adversary can leverage the distribution of outcomes to learn confidential data over time, by invoking this system call many times and observing what value is being returned more frequently than one would expect from a uniform distribution.

Indirect disclosure. Yet another complication with confidentiality is that an adversarial file system might not immediately leak confidential data. For example, an adversarial file system may wait for a legitimate user to read confidential data, at which point the file system would be allowed to access this data, since it has to return it to the user. However, in addition to returning this data, an adversarial file system could also stash away a copy of it, so that the adversary can later retrieve it. For instance, the file system could change the order of entries in an on-disk directory structure, or change the allocated inode numbers or block numbers, based on the confidential data that it wants to leak. Preventing this attack is difficult because the adversarial file system appears to have legitimate access to the user's data when operating on behalf of that user.

File-system complexity. Finally, file systems are complex software. Linux ext4, for instance, consists of approximately 50,000 lines of code. Even the simple verified DFSCQ file system consists of thousands of lines of executable code [13]. The proofs of functional correctness for DFSCQ are already tens of thousands of lines of Coq code. The complexity of proving two-safety, which is a more challenging property, could easily spiral out of control.

5 Specification: data noninterference

To capture the notion of confidentiality in a file system, DISKSEC defines the notion of *data noninterference*. Loosely speaking, data noninterference states that two executions are indistinguishable with respect to specific confidential data (e.g., the contents of a file). Data noninterference allows an application to conclude that an adversary cannot learn the contents of a file from the file system but may be able to learn other information about the file (e.g., its length, its creation time, the fact that it was created at all, etc.). Furthermore, data noninterference does not place any restrictions on application code, which captures the discretionary aspect of typical file-system permissions. This notion intuitively corresponds to the security guarantees provided by Linux file systems.

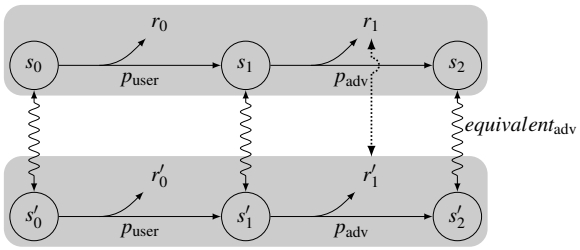


Figure 2: Overview of DISKSEC's approach to reasoning about confidentiality.

Two-safety formulation. DISKSEC formulates data noninterference in terms of two-safety, as shown in Figure 2. Specifically, data noninterference considers two executions that run the same code but start from different states. In Figure 2, the executions are shown as horizontal transitions between states, indicated by the gray outlines. The executions consist of a step by the user (running procedure p_{user}, corresponding to some system call) and then a step by the adversary (running p_{adv}, corresponding to some other system call). Although Figure 2 shows one particular pair of executions, DISKSEC's theorems consider all possible such pairs of executions.

The starting states in these two executions (s_0 and s'_0) agree on all data visible to the adversary but could have different contents of confidential files. We call these two states $equivalent_{\text{adv}}$, to indicate that they are equivalent with respect to the adversary. This equivalence is indicated by the squiggly line in Figure 2. The essence of

data noninterference is allowing the states to differ in the contents of confidential data while requiring all other metadata (such as file length, directory order, etc.) to remain the same.

The definition of data noninterference consists of two requirements. The first is *state noninterference*, which requires that after every transition, the resulting states remain $equivalent_{\text{adv}}$. This is indicated in Figure 2 by the squiggly lines between s_1 and s'_1, as well as between s_2 and s'_2. This requirement ensures that confidential data from s_0 and s'_0 does not suddenly become accessible to the adversary in a subsequent state, and it addresses the indirect-data-disclosure challenge (e.g., an adversarial implementation of the `read` system call stashing away the results).

The second requirement is *return-value noninterference*, which requires that transitions by the adversary return exactly the same values in both executions. For example, Figure 2 shows that the adversary's p_{adv} returns r_1 in the top execution and r'_1 in the bottom execution. Return-value noninterference requires that $r_1 = r'_1$, as indicated by the dotted arrow. This prevents the adversary from learning any confidential data, such as through collusion with an adversarial file system that affects the order of `readdir` results, or through missing access control checks.

Capturing file-system security. Achieving the two requirements from data noninterference ensures that the adversary cannot obtain confidential data from the file system. This is because state noninterference maintains $equivalence_{\text{adv}}$ regardless of what the adversary does (i.e., the squiggly lines will continue to connect states in all possible pairs of executions), and any attempts by the adversary to observe information will produce identical results, based on return-value noninterference, because they run in $equivalent_{\text{adv}}$ states.

The discretionary nature of data noninterference shows up in the fact that legitimate users can obtain different results depending on the confidential data. For example, in Figure 2, the results of the user's execution of p_{user}, r_0 and r'_0, might be different, because p_{user} could correspond to the user reading a confidential file. At this point, a user has the discretion to disclose this information (e.g., by writing it to a public file). Data noninterference does not prevent this, by design, because it is attempting to model the standard discretionary access control in a POSIX file system.

Defining return-value noninterference. Figure 3 presents DISKSEC's definition of return-value noninterference, in a simplified notation. This definition relies on the definition of `exec`, which describes how procedures execute. `exec` takes four arguments: the procedure that is executing (`p`), the principal on whose behalf `p` is running (`u`), the starting state (`st0`), and the randomness for this execution (`rand`). `exec` returns two things: the outcome and an *unseal trace*, which we describe later. The outcome can be either `Finished st' r`, indicating that the procedure ended in state `st'` and returned `r`, or `Crashed st'`, indicating that the system crashed in state `st'`. The unseal traces are irrelevant for now and are used only as part of the proof technique described in §6. This definition also relies on a notion of two states being equivalent for a particular principal, `equivalent_for_principal`, which captures the intuitive notion $equivalent_{\text{adv}}$ from above.

```
Definition equivalent_for_principal u st0 st1 :=
  (* all parts of st0 and st1 that are accessible to
     principal u are identical *).

Definition ret_noninterference '(p : proc T) :=
  forall u st0 st0' rand ret tr0 st1,
    exec p u st0 rand =
        Some (Finished st0' ret, tr0) ->
    equivalent_for_principal u st0 st1 ->
    forall st1' tr1,
      exec p u st1 rand =
          Some (Finished st1' ret', tr1) ->
      ret' = ret.
```

Figure 3: Definition of return-value noninterference, capturing that return values do not leak other users' confidential data.

The definition of return-value noninterference captures the intuition about the adversary not being able to learn information about confidential data: the return value obtained by the adversary by running some code does not depend on the confidential data. To make this precise, `ret_noninterference` of procedure `p` considers pairs of states, `st0` and `st1`, which are equivalent as far as some principal `u` is concerned. Here, `u` is representing the adversary, and confidential data is represented by the difference between `st0` and `st1` that the adversary should not be able to observe. If `u` runs procedure `p` in state `st0` and gets return value `ret`, then it must also have been possible for the adversary to get the same return value, `ret`, if he ran `p` in state `st1` instead.

Defining state noninterference. Figure 4 presents DISKSEC's definition of state noninterference, which complements return-value noninterference. This definition helps DISKSEC deal with the indirect-disclosure challenge from §4.2. This definition considers two principals: a `viewer` and a `caller`. The definition intuitively says that, by running procedure `p`, the caller will not create any state differences observable to `viewer`.

More formally, `state_noninterference` considers two executions by `caller`, running the same procedure `p`, with the same exact arguments (encoded inside of `p`). If the caller runs `p` in two states that appear equivalent to `viewer`, then the resulting states in `res0` and `res1` will still appear equivalent to `viewer`. This definition includes the possibility of a crash while running `p`.

```
Definition equiv_state_for_principal u res0 res1 :=
  exists st0 st1,
    equivalent_for_principal u st0 st1 /\
    (res0 = Crashed st0 /\ res1 = Crashed st1 \/
     exists v0 v1,
       res0 = Finished st0 v0 /\
       res1 = Finished st1 v1).

Definition state_noninterference `(p : proc T) :=
  forall viewer caller st0 rand res0 tr0 st1,
    exec p caller st0 rand = Some (res0, tr0) ->
    equivalent_for_principal viewer st0 st1 ->
    forall res1 tr1,
      exec p caller st1 rand = Some (res1, tr1) ->
      equiv_state_for_principal viewer res0 res1.
```

Figure 4: Definition of state noninterference, capturing that `caller` does not indirectly disclose state to `viewer`.

Handling non-determinism and probabilities. Both Figure 3 and Figure 4 quantify over an argument called `rand` that is passed to `exec`. `rand` is an oracle that supplies all non-determinism used during execution, including non-deterministic values used by the file system implementation (e.g., getting a random number), as well as non-determinism representing the effect of a crash (i.e., the point at which the crash occurred, and which recent writes made it to disk). The execution semantics, `exec`, queries the `rand` oracle whenever it needs to make a non-deterministic decision. The `exec` function is deterministic given a specific `rand`, but DISKSEC allows non-determinism by permitting different executions with different non-determinism oracles. One way to think of this `rand` oracle is that it represents a seed for a logical random-number generator.

Factoring out the randomness `rand` from the execution semantics `exec` helps DISKSEC handle probabilities without fully formalizing probabilistic reasoning in Coq. Since the `exec` function is deterministic (given a specific randomness oracle `rand`), the probability of a particular outcome is the sum of the probabilities of different `rand` oracles that lead to that outcome. Following the random-number generator seed analogy, the probability of an outcome is simply the fraction of seeds that lead to that outcome.

DISKSEC's theorem statements require that, for any choice of `rand`, both the return values and states are equivalent. This ensures that the probabilities of equivalent return values and states are also equal, since the probabilities of these outcomes are simply the sums of probabilities of individual `rand` values. Using the samples to relate the probabilities of outcomes is reminiscent of a coupling argument [21], although we do not explicitly reason about probabilities in DISKSEC. Since the probabilities of the outcomes are equal, this prevents an adversary from learning confidential data based on the observed probabilities of different outcomes.

6 Proof approach: sealed blocks

Proving that every system call in a file system satisfies `ret_noninterference` and `state_noninterference` would require a proof that reasons about two executions, which is complex. To reduce proof effort, DISKSEC introduces an implementation and proof approach called *sealed blocks*. This approach factors out reasoning about confidentiality of files from most of the file-system logic, by reasoning about the confidentiality of disk blocks. The intuition behind this approach is threefold. First, all confidential data lives in file blocks. Second, the file system itself rarely needs to look inside of the file blocks. Finally, permissions on files translate directly into permissions on the underlying blocks comprising the file.

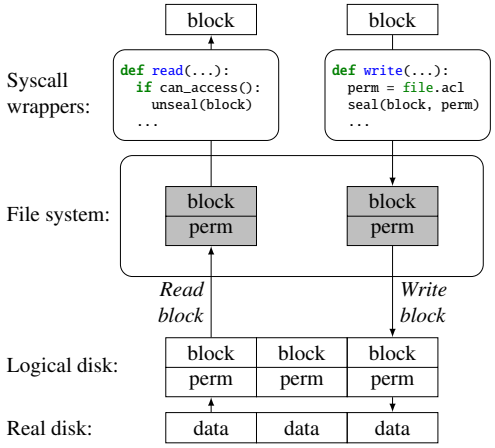


Figure 5: Overview of DISKSEC's proof approach using sealed blocks.

Figure 5 presents an overview of DISKSEC's block-sealing approach. There are three parts to the block-sealing approach. The first is to create a logical disk where every disk block is associated with a *permission*, which defines the set of principals that can access this block. Some permissions are public, indicating that the block is accessible to anyone. Other permissions might restrict access to some users, indicating that this block is storing confidential file data. DISKSEC is agnostic to the specific choice of principals or permissions; that is, all of DISKSEC is parameterized over arbitrary types for principals and permissions. The logical disk is purely a proof strategy and does not appear at runtime; the real disk, shown at the bottom of Figure 5, has no permissions.

The second part is a sealed-block abstraction, indicated by shaded blocks in Figure 5. A sealed block represents the raw block contents and the associated permission, but the file system cannot directly access a sealed block's contents. Instead, the file-system implementation must explicitly call `seal()` and `unseal()` to translate between sealed blocks and their raw contents. These `seal()` and `unseal()` functions are also purely part of the proof and do not appear at runtime.

The code of the file system can read and write arbitrary blocks on disk, but the result of a read is a sealed block that must be explicitly unsealed if needed. The file-system internals can unseal public blocks (e.g., containing allocator bitmaps or inodes) but cannot unseal private blocks. This avoids the need to reason about the file-system implementation when proving confidentiality, because the file-system implementation never has access to confidential data.

The third part is the wrappers for system calls that handle confidential data, namely, `read()` and `write()`. These wrappers are responsible for explicitly calling `seal()` and `unseal()` to translate between the raw data seen by the user (on top of the system call) and the sealed blocks that are handled in the rest of the file-system implementation.

DISKSEC's sealed-block approach is a good fit for the challenges outlined in §4.2. Specifically, there are very few places where a file system must access the actual contents of a file's disk block—namely, in the wrappers for the `read()` and `write()` syscalls. As a result, most specifications in a file system remain largely the same. The key difference is that the specifications promise that the procedure in question does not look inside of any confidential blocks. This means that any nondeterminism present in the specification cannot be used to leak confidential data.

This approach allows file-system developers to avoid proving explicit confidentiality theorems for most of the file system, but it still allows DISKSEC to conclude that confidentiality is not violated. DISKSEC provides a theorem that proves two-safety for any file-system implementation that correctly uses the sealed-block abstraction. As a result, the file-system developer need not reason about complex two-safety theorems and can limit their reasoning to single executions.

6.1 Formalizing sealed blocks

To formally define DISKSEC's sealed-block abstraction, DISKSEC uses the notion of a *handle* to represent a sealed block. DISKSEC requires the developer to perform two steps. The first is to modify their code to use the sealed-block abstraction: that is, to pass around handles for blocks and to call `seal()` and `unseal()` as necessary. The second is to prove that their code correctly follows the unsealing rules. This boils down to ensuring that sealed blocks are unsealed only when the principal has appropriate permission for that block.

DISKSEC models this by extending traditional Hoare logic to reason about unseal operations. Specifically, DISKSEC builds on CHL [12], where functional correctness specifications are written in terms of pre- and post-conditions. DISKSEC, first, extends the execution semantics (as we describe next) to produce an *unseal trace* consisting of unseal operations and, second, extends the specifications to require that the unseal trace contain only allowed unseals.

We expect that systems built on top of DISKSEC would often group multiple blocks into a single object (e.g., multiple blocks comprising a single file in a file system). To help developers reason about all of these blocks sharing the same permissions, DISKSEC introduces the notion of a *domain*. This is a layer of indirection between blocks and permissions. Specifically, sealed blocks point to a domain ID (e.g., an inode number in the case of a file system), and the domain in turn specifies the permission for those blocks (e.g., the permission reflected in the inode's data structure).

Execution model. DISKSEC's execution model requires the implementation to be written in a domain-specific language, based on CHL and implemented inside of Coq, which provides several primitive operations. These operations include reading and writing the disk, manipulating sealed blocks by sealing and unsealing, as well as others for sequencing computation, returning values, flushing disk writes, etc.

Figure 6 shows a simplified version of DISKSEC's execution semantics. The semantics are defined as a function that takes the `code` being executed (of type `proc T`), the principal `u` running the operation (of type `Principal`), the starting state `st` (of type `State`), and a randomness oracle `rand`. The function produces a tuple consisting of a result (of type `result T`) and a trace of unsealed permissions (of type `trace`). The function is allowed to return `None` (as indicated by the `option` type) when there is no execution possible for the supplied randomness (e.g., the randomly chosen handle is already in use).

For example, consider the case that handles the `Read a` operation, which describes the execution of reading address `a` from disk. There are three sub-cases. If the address is out of bounds, the `Read` returns a handle for a zero block, with an empty unseal trace. If the handle `h` supplied by the randomness oracle is already in use, no execution is possible. Otherwise, the `Read` initializes the new handle to represent the block from address `a`, with the block's domain ID, and returns that handle, with an empty trace because no blocks were unsealed.

As another example, the `Unseal h` operation produces a nonempty trace, consisting of the permission of the sealed block whose handle `h` was unsealed, as long as the handle was valid (otherwise, `Unseal` returns zero). Since the sealed block points to a domain ID, `dom`, the semantics of `Unseal` look up the corresponding permissions of that domain. One omitted rule handles concatenation of unseal traces when a developer sequences one statement after another.

The `ChangePerm dom newperm` operation allows the developer to change permissions of a domain. This operation is used in implementing `chown`. The semantics of

```
Inductive nondet_decision :=
| RandomHandle (h:handle)
(* Other types of non-determinism omitted for space *)
| CrashHere.

Definition oracle := list nondet_decision.

Definition exec '(code:proc T) (u:Principal) (st:State)
           (rand:oracle) : option (result T * trace) :=
  match code, rand with
  | ChangePerm _ _, CrashHere => None
  | _, CrashHere => Some (Crashed st, [])
  | Read a, RandomHandle h =>
    if addr_out_of_bounds st a then
      Some (Finished st hzero, [])
    else if handle_used st h then
      None
    else
      let data := disk_block_data st a in
      let dom  := disk_block_dom  st a in
      let st' := install_handle st h (data, dom) in
      Some (Finished st' h, [])
  | Write a h, _ =>
    if handle_used st h then
      let data := handle_data st h in
      let dom  := handle_dom  st h in
      let st' := disk_block_write st a (data, dom) in
      Some (Finished st' tt, [])
    else
      Some (Finished st tt, [])
  (* Some transitions omitted for space reasons *)
  | Seal data dom, RandomHandle h =>
    if handle_used st h then
      None
    else
      let st' := install_handle st h (data, dom) in
      Some (Finished st' h, [])
  | Unseal h, _ =>
    if handle_used st h then
      let data := handle_data st h in
      let dom  := handle_dom  st h in
      let perm := domain_perm st dom in
      Some (Finished st data, [perm])
    else
      Some (Finished st zero, [])
  | ChangePerm dom newperm, _ =>
    let oldperm := domain_perm st dom in
    let st' := domain_set_perm st dom newperm in
    Some (Finished st tt, [oldperm])
  | _, _ => None
  end.
```

Figure 6: Execution semantics with logging of unseal operations.

`ChangePerm` modify the permission associated with the domain, and produce an unseal trace containing the domain's old permission, to reflect that data with that permission may have been disclosed. Since the domains are purely a proof construct, `ChangePerm` is a purely logical operation, which does not perform any actions at runtime.

Finally, `exec` describes the possible crash behaviors of the system. For example, the case for `_, CrashHere` states that it is possible to crash in the starting state, regardless of what code was being executed, if the randomness oracle tells us `CrashHere`. A combination of other rules, not shown, allow crashing in the middle of a sequence of operations. The very first case, for `ChangePerm _ _, CrashHere`, says that `ChangePerm` cannot crash. This reflects the fact that `ChangePerm` is a purely logical operation.

Specification and verification of unseal rules. DISKSEC requires developers to write specifications for each procedure, using pre- and postconditions. The postcondition describes how the procedure modifies the state of the system, along with what must be true of the procedure's return value, assuming that the precondition (a predicate over the system state and the procedure's arguments) held at the start of the procedure.

To reason about what blocks a procedure might unseal, DISKSEC augments specification postconditions with requirements about the permissions that appear in the unseal trace produced by the execution of the procedure.

```
Definition unseal_safe '(p : proc T) :=
  forall u st rand res tr,
    exec p u st rand = Some (res, tr) ->
      forall perm,
        In perm tr -> can_access u perm.
```

Figure 7: Definition of unseal safety.

Figure 7 shows DISKSEC's definition of unseal safety. This definition says that procedure `p` is "unseal-safe" if, for every principal `u` that runs this procedure and any starting state `st`, all permissions produced by this procedure in its unseal trace `tr` will be accessible to the calling principal. Proving unseal safety leads to a proof obligation for the file-system developer—namely, proving that the implementation will unseal a block only if the current principal has access to it.

File-system implementation code falls into three categories with respect to proving unseal safety. The first category are procedures that do not invoke any `Unseal` operations. For these procedures, the resulting unseal trace is always empty, and DISKSEC is able to prove unseal safety without any developer input. Most of the file-system code falls in this category.

The second category are procedures that unseal public blocks. Examples include accessing inodes, allocator bitmaps, directories, etc. These procedures do produce unseal traces containing permissions, but all of the permissions should be public. Thus, the developer's job is to show that these permissions are indeed public; once this is established, showing that the current principal has access is straightforward (since every principal has access to public permissions).

To prove that the permissions are indeed public, the developer relies on representation invariants of the file system. For example, the invariant for the block allocator states that all of the bitmap blocks are public. The developer can assume this invariant within any implementation of the block allocator API, which helps her prove that

the block in question has public permissions. In turn the developer must prove that the invariant is preserved by every procedure (including across crashes and recovery), and show that it is established at initialization time by `mkfs`.

The final category are procedures that unseal private blocks. In a file system, this happens only in the implementation of the `read` system call, which returns file data to the caller. The implementation (wrapper) of the `read` system call contains explicit code to obtain the current principal, get the file's ACL (access control list) from the inode, and compare them. The developer's job is to prove that this code correctly performs the permission check. This proof typically relies on the file's representation invariant, which asserts that every file block is tagged with a permission matching the ACL stored in the inode.

```
Definition unseal_public `(p : proc T) :=
  forall u st rand res tr,
    exec p u st rand = Some (res, tr) ->
      forall perm,
        In perm tr -> perm = Public.
```

Figure 8: Definition of `unseal_public`.

DISKSEC also provides a stronger version of unseal-safety, as shown in Figure 8, called `unseal_public`. A procedure satisfies this definition if all of its code falls in the first two categories above: that is, the procedure either unseals no blocks at all or unseals only public blocks. This alternative definition is strictly stronger than unseal-safety; any procedure that satisfies `unseal_public` is also unseal-safe. The distinction between these two notions will help the developer prove noninterference theorems, as we will describe in §6.2.

Crashes. DISKSEC's approach naturally extends to reasoning about crashes. DISKSEC's disk-crash model builds on the CHL model of disk crashes [12, 13]. After a crash, disk blocks can be updated nondeterministically, as in CHL, based on outstanding writes that are in the disk's write buffer but have not been flushed yet to durable storage. However, domains always follow the data for pending writes; that is, logically, the content of the disk block is updated atomically together with its domain ID.

All handles are invalidated after a crash, to model the fact that the computer reboots and all in-memory state is lost. All recovery code, such as log replay or `fsck`, is proven correct in DISKSEC, which means that it must follow the same block-sealing rules as the rest of the file-system code. This ensures that no data can be disclosed by the recovery code.

6.2 Proving noninterference

To help the developer prove the two types of noninterference, DISKSEC provides helper theorems. Figure 9 shows the first one, which proves return-value noninterference based on unseal-safety. DISKSEC proves this theorem by considering all operations performed by procedure `p`. Each operation must produce the same result in the two executions being considered, since the states are equivalent for the principal in question, `u`. The only way in which the executions could differ is if they unsealed a block that was not accessible to `u`. However, `unseal_safe` says that this is impossible. This theorem also applies to procedures that are `unseal_public`, since that notion is strictly stronger than `unseal_safe`.

```
Theorem unseal_safe_to_ret_noninterference :
  forall `(p : proc T),
    unseal_safe p -> ret_noninterference p.
```

Figure 9: Theorem connecting unseal-safety to return-value noninterference.

Figure 10 shows the second theorem provided by DISKSEC, for reasoning about state noninterference. This theorem requires that the procedure satisfy the stronger definition, `unseal_public`, to ensure state noninterference. The intuition for why this theorem is true lies in the fact that a procedure that unseals only public blocks cannot obtain any confidential data in the first place. As a result, this procedure's execution will be identical regardless of the contents of confidential blocks, and thus the state after this procedure's execution will remain equivalent from the adversary's point of view. DISKSEC proves this theorem formally in Coq.

```
Theorem unseal_public_to_state_noninterference :
  forall `(p : proc T),
    unseal_public p -> state_noninterference p.
```

Figure 10: Theorem connecting `unseal_public` to state noninterference.

DISKSEC does not provide a general-purpose theorem for reasoning about state noninterference for procedures that satisfy only the weaker notion of unseal-safety (i.e., that unseal private blocks), such as the `read()` system call. Such procedures can indirectly disclose data as described in §4.2 to legitimately unseal confidential data on behalf of the currently executing principal but then stash a copy of it. It is up to the file-system developer to prove the state noninterference of those procedures. §7 will discuss in more detail how SFSCQ structures its implementation to simplify these proofs; in the case of SFSCQ, the only system call that requires this type of reasoning is `read`.

6.3 Code generation

To generate efficient executable code, DISKSEC must avoid explicitly sealing and unsealing blocks. To do so, DISKSEC eliminates any notion of handles, sealing, or unsealing at runtime. DISKSEC does so by representing each handle with the actual disk-block contents themselves, when generating executable code. DISKSEC's theorems ensure that the code does not look at the disk contents at runtime unless it has the appropriate permissions. As a result, it is safe to perform this elimination.

Similarly, this allows the sealing and unsealing operations also to be eliminated from runtime code.

7 Case study: File system

To evaluate whether DISKSEC allows specifying and proving confidentiality for a file system, we applied DISKSEC to the DFSCQ verified file system, producing the SFSCQ verified secure file system, as described below.

7.1 Specifying security

The core specification of confidentiality for SFSCQ lies in the `write` system call, as shown in Figure 11. This specification says that the *data* argument to the `write` system call remains confidential. This is stated formally by considering two different executions, starting from the same state `st`, where different data (`data0` and `data1`) are written to the same offset `off` of the same file `f`. The results, `res0` and `res1`, must be equivalent for any adversary `adv` that does not have permission to access file `f`. Since `equivalent_state_for_principal` considers both crashing and noncrashing executions, this definition ensures that the data passed to `write` remains confidential regardless of whether the system crashes or not.

```
Theorem write_confidentiality :
  forall f off data0 data1 caller st rand res0 tr0,
    exec (write f off data0) caller st rand =
                              Some (res0, tr0) ->
      exists res1 tr1,
        exec (write f off data1) caller st rand =
                              Some (res1, tr1) /\
        forall adv,
          ~ can_access adv (file_perm st f) ->
          equiv_state_for_principal adv res0 res1.
```

Figure 11: Confidentiality specification for the `write` system call.

The other part of the security specification lies in the `chown` system call, which changes the permissions on existing files, and thus affects what data is or is not confidential. Because `chown` can disclose the contents of a previously confidential file, the standard definition of state non-interference from Figure 4 does not hold for `chown`. Specifically, even if an adversary `viewer` could not distinguish states `st0` and `st1` before some `caller` executed `chown`, the adversary may nonetheless be able to distinguish `st0` and `st1` after the `chown` runs because the adversary may now have permission to read the previously confidential file.

The security of `chown` is defined by a specialized version of state non-interference, which considers three cases. The first case is that the adversary `viewer` does not have access to the file after the `chown` (i.e., is not the new owner). In this case, state non-interference holds. The second case is that the adversary `viewer` does gain access to the file after `chown` (i.e., is the new owner), but the file had the same contents in the two executions (i.e., in states `st0` and `st1`). In this case, state non-interference holds as well. Finally, the adversary `viewer` may gain access to the file *and* the files had different contents in the two executions. In this case, state non-interference does not apply. Figure 12 summarizes this formally.

```
Definition chown_state_noninterference f new_owner :=
  forall viewer caller st0 rand res0 tr0 st1,
    exec (chown f new_owner) caller st0 rand =
                                Some (res0, tr0) ->
    ( file_data st0 f = file_data st1 f \/
      viewer <> new_owner ) ->
    equivalent_for_principal viewer st0 st1 ->
      exists res1 tr1,
        exec (chown f new_owner) caller st1 rand =
                                Some (res1, tr1) /\
        equiv_state_for_principal viewer res0 res1.
```

Figure 12: Confidentiality specification for the `chown` system call.

The `write` and `chown` specifications, shown above, are the only parts of the security specification that are specific to the file system, because they define where confidential data enters the system in the first place, and how permissions on that confidential data can change. Somewhat counter-intuitively, no special treatment is required in the specifications of other system calls, such as `read`. Instead, it suffices to prove the two general noninterference theorems for all system calls (i.e., `ret_noninterference` and `state_noninterference`). This is because we do not want to consider specific attacks, such as whether `read` has a missing access-control check. Instead, DISKSEC's noninterference definitions ensure that confidential data cannot be disclosed regardless of what system calls the adversary tries to use.

Integrity of the file system is a functional-correctness property and thus is covered by SFSCQ's specifications, alongside other correctness properties. Integrity did not require SFSCQ to use any machinery from DISKSEC for reasoning about confidential data.

7.2 Modifying the implementation

Changing representation invariants. DFSCQ consists of many modules, such as the write-ahead log, the bitmap allocator, the inode module, etc. Each module has its own invariant that describes how that module's state is represented in terms of blocks. For example, the bitmap allocator describes how the free bits are packed into disk blocks, where they are stored on disk, and the semantics of each bit.

For SFSCQ, we modified all invariants that describe disk blocks to state the domain IDs that go along with those blocks. For instance, we modified the invariant of the allocator to state that the bitmap blocks are public. We modified the write-ahead log layer to expose the underlying domain IDs on disk blocks to modules implemented on top of the write-ahead log (in addition to modifying the log invariant to state that the log metadata is public).

The only nonpublic data is the file contents. We modified the file invariant to state that the domain ID of every file block matches the file's inode number, and the permissions for a particular domain ID match the ACL stored in the inode with the inode number matching the domain ID.

One surprising issue that we encountered came up in the DFSCQ write-ahead log. For performance, DFSCQ's write-ahead log used checksums to verify block contents after a crash. As a result, the recovery procedure unsealed blocks from the write-ahead log after a crash, including blocks that contain confidential data.

To address this issue, we switched to a barrier-based write-ahead log instead, which is the default design of Linux ext4. Instead of using checksums, the barrier-based write-ahead log issues a disk flush between writing the contents of new log entries and updating the log header. (DFSCQ already included an implementation of this barrier-based write-ahead log but did not use it by default.)

Modifying code. Loosely speaking, DFSCQ modules handle two kinds of blocks: blocks that they manipulate (e.g., the bitmap allocator manipulating the bitmap blocks) and blocks that they pass through (e.g., the write-ahead log handling reads and writes as part of a transaction, or the file layer handling file reads and writes). The first category required a module to access the block contents, so we added `Seal` and `Unseal` operations accordingly. Virtually all operations that fell in this category involved sealing and unsealing public data. For the second category, we did not seal or unseal the data and instead transparently passed through the handle representing the block; as a result, the module was oblivious to the domain IDs associated with the disk block.

Private data is sealed and unsealed at the top of the SFSCQ implementation; that is, in the implementation of the `read` and `write` system calls. We modified the `write` system-call implementation to `Seal` the blocks with the file's inode number as the domain ID, before processing them further. We modified the `read` system call to implement the permission-checking logic—i.e., reading the ACL from the file's inode, checking whether the currently running principal has access to the file, and unsealing the block only if the check passes.

Changing intermediate specifications. We augmented the Hoare-logic specifications of all internal SFSCQ procedures to require that the procedure be `unseal_public`. This change required little manual effort, because we simply changed the underlying definition of the Hoare-logic specification to require `unseal_public`. For the write-ahead log, we added additional constraints in the specification of the `log_write` procedure, requiring that the blocks written as part of a transaction must be public, as described above.

7.3 Proving security

Reproving functional correctness. Many existing proofs in DFSCQ broke after we made the above changes. The proofs broke for three reasons: there were now additional `Seal` and `Unseal` operations in the code (e.g., the bitmap allocator now sealed and unsealed its bitmap blocks), the logical representation of a block changed to include a domain ID, and the specification changed (e.g., augmenting the invariant to state the domain ID of a block). This required manually tweaking most of the proofs to fix them. The proof changes were simple since the code's logic and the proof argument remained unchanged.

Proving unsealing. In addition to fixing existing proofs, SFSCQ's specifications required us to prove that the `Unseal` operation was used correctly. For most procedures, the specification required that the procedure satisfy `unseal_public`. Proving that only public blocks were unsealed required us to demonstrate that the block was indeed public by referring to the invariant.

For the implementation of the `read` system call, which unseals private data, we had to prove that `read` correctly implements the permission check in its code. This means proving that `read` calls `Unseal` only after checking permissions, and that the code for the permission check returns "allowed" only if the current principal really does have permission to access the file contents. This proof mostly boiled down to showing that the code implementing the access-control check in `read` matches the logical permission required by the specification.

Proving noninterference. Proving that SFSCQ provides confidentiality required us to prove three theorems. The first is that `write` implements the specification from Figure 11. This shows that SFSCQ will treat data passed by an application to `write` as confidential. The second is that system calls satisfy `ret_noninterference`. This shows that an adversary cannot use any of SFSCQ's system calls to learn confidential data. The final is that all system calls satisfy `state_noninterference`. This shows that SFSCQ will not indirectly leak a user's data when the user invokes an otherwise-benign system call. Taken together, these theorems allow an application to formally conclude that its data remains private, as we show in §9.

Proving `ret_noninterference` was the easiest, using DISKSEC's theorem from Figure 9. All SFSCQ procedures are proven to be unseal safe, so no further proof effort is required.

Proving `state_noninterference` was simple for all system calls except `read`, because those system calls satisfy `unseal_public`, allowing us to apply DISKSEC's theorem from Figure 10. For `read`, we structured the

system-call implementation in two parts: a `read_helper`, which returns the handle to the data read from the file, and a wrapper around `read_helper` that unseals the data and returns it to the user. `read_helper` is `unseal_public`, allowing us to apply DISKSEC's theorem from Figure 10. The wrapper required a manual proof, but the proof was short since the wrapper is two lines of code.

Finally, to prove that `write` meets its confidentiality specification, we similarly split `write` into a wrapper and a `write_helper`. The wrapper's job is to seal all input data and pass the handles to `write_helper`. Much as with `read`, this reduced the proof effort to just the wrapper.

8 Implementation

We implemented DISKSEC by extending the CHL framework from FSCQ [12]. The changes involved modifying the model of the disk to keep track of logical permissions, adding primitive operations to seal and unseal blocks, and changing the execution semantics to keep track of unseal permissions, as shown in Figure 6. We also changed the meaning of Hoare-logic specifications to require either unseal-safety or the stronger `unseal_public` notion. The source code of DISKSEC and SFSCQ is publicly available at https://github.com/mit-pdos/fscq.

We developed SFSCQ by modifying the DFSCQ file system [13], making the changes described in §7. In particular, as mentioned in §7.2, we switched from DFSCQ's checksum-based write-ahead log to a two-barrier-based log in SFSCQ (which is also the default for Linux ext4). SFSCQ retains all other optimizations from DFSCQ, including log-bypass writes, deferred commit, etc (with proofs). As with DFSCQ, we produce executable code by extracting the Coq implementation to Haskell and running it on top of FUSE. To erase the block sealing and unsealing operations at runtime, DISKSEC uses the `Extract Constant` command in Coq to represent DISKSEC's handles using the raw blocks themselves, and it implements `Seal` and `Unseal` as no-ops.

We built two versions of DISKSEC and SFSCQ. The first version is fully proven, but lacks support for changing permissions on an existing file (i.e., changing the permissions on a file would require copying the file's data into a new file with the new permissions), and lacks support for randomness oracles. The second version extends the first version with support for randomness oracles and dynamic permissions. These changes caused existing proofs to break, and a few of them have not been repaired. See the source code for details.

The DISKSEC approach worked reasonably well for SFSCQ because the underlying FSCQ file system does not unseal user data unless the user explicitly reads it. The one exception was in the checksum-based write-ahead log, as mentioned above. Other file system features that look at file contents might also be a challenge for DISKSEC, such as proactive checksum verification of file contents, de-duplication, storing small file contents in the inode itself, etc.

9 Evaluation

This section experimentally answers the following questions:

- Are SFSCQ's specifications trustworthy? That is, are SFSCQ's theorems sufficient for applications to prove confidentiality of their own data? What assumptions do these proofs rely on?
- How much effort was required to develop DISKSEC, and to use DISKSEC to prove the security of SFSCQ?
- How much runtime overhead does DISKSEC's approach impose in SFSCQ?

9.1 Specification trustworthiness

To evaluate the trustworthiness of SFSCQ's specifications, we performed several analyses.

End-to-end application confidentiality. To demonstrate that SFSCQ's specifications capture confidentiality in a useful way, we developed a simple application on top of SFSCQ that copies a file, wrote a confidentiality specification for this application (namely, that the application does not leak the data of the copied file), and proved it. This application tests two aspects of SFSCQ's specs. The first is, does SFSCQ's specification actually guarantee confidentiality? The second has to do with SFSCQ's discretionary access control model: can application developers demonstrate that they are not inadvertently leaking data, despite having the discretion to do so?

We were able to prove the correctness and security of our implementation of `cp`. This suggests that SFSCQ's specifications capture sufficient information for `cp` to conclude that its data remains confidential, and that it is possible for application developers to show that they do not abuse their discretionary privileges by leaking data.

Bug case study. To evaluate whether SFSCQ's specifications would eliminate real security bugs, we qualitatively analyzed the bugs presented in §3 to determine whether SFSCQ's theorems preclude the possibility of that bug. Figure 13 shows the results. Functional correctness theorems preclude the possibility of integrity bugs. DISKSEC state noninterference precludes the possibility of all confidentiality bugs in our study. No bugs were prevented by return-value noninterference, because return-value noninterference captures a particularly simple kind of bug, such as the file system forgetting to check the ACL on `open()`. No file-system developers made this mistake in our study. Nonetheless, return noninterference is important for completeness of SFSCQ's theorems. Overall, the results demonstrate that SFSCQ's theorems preclude the possibility of all studied bugs.

Description	Theorem violated
anyone can change POSIX ACLs [3, 5, 8]	state NI
reiserfs permissions can be changed by writing to hidden file [2]	state NI
truncated data can be accessed [7]	state NI
crash can expose deleted data in ext4 [9]	state NI
crash can expose data in ext4 [22]	state NI
can overwrite append-only file in ext4, btrfs [4, 6]	integrity
can overwrite arbitrary files in ext4 [1]	integrity

Figure 13: Security bugs in Linux file systems and which SFSCQ theorem precludes them.

Trusted computing base. SFSCQ assumes the correctness of several components. SFSCQ assumes that Coq's proof checking kernel is correct, because it verifies SFSCQ's proofs. SFSCQ assumes that the Haskell runtime and support libraries (and the underlying Linux kernel) do not have bugs, since SFSCQ generates executable code through extraction to Haskell. SFSCQ assumes that DISKSEC's model of the disk is accurate. In particular, all non-determinism in DISKSEC's execution semantics must be "realizable," in the sense that it is actually possible for an execution to observe all specified non-determinism (e.g., crashing at any point), and this non-determinism must be independent of confidential data. All proofs in DISKSEC and SFSCQ are checked by Coq.

9.2 Effort

To understand how much effort was required to verify DISKSEC and SFSCQ, we compared SFSCQ to the implementation of DFSCQ on which SFSCQ is based. Figure 14 shows the results (counting the sum of lines removed and lines added), breaking down the differences into several categories. The core infrastructure, including improvements to DFSCQ's CHL, amounted to around 9,300 lines. We made significant changes to DFSCQ to develop SFSCQ, but many of these changes were mechanical fixes to proofs to address small changes. In addition, using DISKSEC in SFSCQ required around 1,900 lines of new code and proofs. Porting DFSCQ to the first version of DISKSEC (without support for changing permissions) took one author about 3 months, and another 2 months to mostly finish support for permission changes.

Component	Changes to DFSCQ
DISKSEC	9,283
DFSCQ proof fixes	$-10,471$, $+26,433$ (36,094 total)
SFSCQ impl. and proofs	1,837
Verified `cp` application	407

Figure 14: Lines of code change required to implement DISKSEC and apply it to build SFSCQ. Counts measure the diff between DFSCQ and SFSCQ.

9.3 Performance

We expect that the performance overhead of DISKSEC is nearly zero, because most of its code changes (such as handles, sealing, and unsealing) are eliminated in the process of generating executable code. (All of the Seal and Unseal operations turn into `return` statements.) The only exception is checking permissions when reading data from a file; the original DFSCQ implementation had no permission checks, which we added in SFSCQ.

To check that DISKSEC introduces almost no overhead, we used two microbenchmarks (LFS smallfile and largefile benchmarks [31] as modified by DFSCQ [13]). As a baseline, we compare with two versions of DFSCQ, on which SFSCQ is based. The first is unmodified DFSCQ. The second is a version of DFSCQ with a two-disk-barrier write-ahead log (instead of its default checksum-based log). This matches the modification we made to SFSCQ, as mentioned in §7.2. For comparison with other file systems, such as Linux ext4, we refer the reader to the detailed evaluation in the DFSCQ paper [13: §7.4].

Figure 15 shows the results, which confirm that SFSCQ performs nearly identically to DFSCQ in the same logging configuration. The use of a two-disk-barrier write-ahead log incurs some performance overhead for smallfile; largefile performance is not impacted because its file data writes bypass the log.

Filesystem	smallfile	largefile
DFSCQ	446 files/s	108 MB/s
DFSCQ (no checksums)	295 files/s	109 MB/s
SFSCQ	299 files/s	100 MB/s

Figure 15: Benchmarks showing performance of SFSCQ compared to DFSCQ and a version of DFSCQ with a comparable logging implementation. Numbers shown are the median of 30 runs.

10 Conclusion

SFSCQ is the first file system with a machine-checked proof of security. DISKSEC enabled us to specify and prove SFSCQ's confidentiality with modest effort. DISKSEC's key techniques are the use of a sealed block abstraction, as well as the notion of data noninterference as the top-level theorem statement, which is a good fit for discretionary file access control. Experimental evaluation shows that SFSCQ's theorems would preclude security bugs that have been found in real file systems, that SFSCQ's development effort was moderate, and that there is little performance impact of using DISKSEC.

Acknowledgments

Thanks to the PDOS group, the anonymous reviewers, and to our shepherd Jay Lorch, for improving this paper. This research was supported by NSF awards CNS-1563763 and CNS-1812522, and by Google.

References

[1] CVE-2009-4131, 2009. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4131.

[2] CVE-2010-1146, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[3] CVE-2010-2017, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1641.

[4] CVE-2010-2066, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[5] CVE-2010-2017, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2071.

[6] CVE-2010-2537, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[7] CVE-2015-8374, 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8374.

[8] CVE-2016-1237, 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[9] CVE-2017-7495, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7495.

[10] J. Barkley. Introduction to POSIX security, Oct. 1994. http://ftp.gnome.org/mirror/archive/ftp.sunet.se/pub/security/docs/nistpubs/800-7/node18.html.

[11] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[13] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, Oct. 2017.

[14] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–118, Vancouver, Canada, Oct. 2010.

[15] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, Santa Barbara, CA, June 2016.

[16] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 350–357, Nashville, TN, Oct. 1981.

[17] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, Oct. 2017.

[18] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 186–197, Venice, Italy, Jan. 2004.

[19] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, Apr. 1982.

[20] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[21] J. Hsu. *Probabilistic Couplings for Probabilistic Reasoning*. PhD thesis, University of Pennsylvania, Nov. 2017.

[22] J. Kara. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977, Nov. 2014.

[23] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM*

Transactions on Computer Systems, 32(1):2:1–70, Feb. 2014.

[24] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 158–170, Long Beach, CA, Jan. 2005.

[25] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–57, Jan. 1992.

[26] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[27] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–147, Saint-Malo, France, Oct. 1997.

[28] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, Oct. 2000.

[29] A. Popescu, J. Hölzl, and T. Nipkow. Proving concurrent noninterference. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP)*, pages 109–125, Kyoto, Japan, Dec. 2012.

[30] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 16th IEEE Symposium on Security and Privacy*, pages 114–127, Oakland, CA, May 1995.

[31] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[32] J. M. Rushby. Proof of separability: A verification technique for a class of security kernels. In *Proceedings of the 5th International Symposium on Programming*, pages 352–367, Turin, Italy, Apr. 1982.

[33] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 78–94, Cambridge, MA, June 2012.

[34] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*, pages 352–367, London, UK, Sept. 2005.

[35] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*, Apr. 2018. URL https://doi.org/10.5281/zenodo.1219885.

[36] T. Ts'o. [PATCH] ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. http://lists.openwall.net/linux-ext4/2012/11/16/9, Nov. 2012.

[37] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, Jan. 2012.

[38] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 631–647, Santa Barbara, CA, June 2016.

[39] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[40] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

Proving the correct execution of concurrent services in zero-knowledge

Srinath Setty[⋆], Sebastian Angel[⋆◇], Trinabh Gupta[⋆†], and Jonathan Lee[⋆]
[⋆]*Microsoft Research* [◇]*University of Pennsylvania* [†]*UCSB*

Abstract. This paper introduces Spice, a system for building *verifiable state machines (VSMs)*. A VSM is a request-processing service that produces proofs establishing that requests were executed correctly according to a specification. Such proofs are *succinct* (a verifier can check them efficiently without reexecution) and *zero-knowledge* (a verifier learns nothing about the content of the requests, responses, or the internal state of the service). Recent systems for proving the correct execution of stateful computations—Pantry [25], Geppetto [34], CTV [30], vSQL [83], etc.—implicitly implement VSMs, but they incur prohibitive costs. Spice reduces these costs significantly with a new storage primitive. More notably, Spice's storage primitive supports multiple writers, making Spice the first system that can succinctly prove the correct execution of concurrent services. We find that Spice running on a cluster of 16 servers achieves 488–1167 transactions/second for a variety of applications including inter-bank transactions [27], cloud-hosted ledgers [28], and dark pools [63]. This represents an 18,000–685,000× higher throughput than prior work.

1 Introduction

We are interested in a system for building *verifiable state machines (VSMs)*. A VSM is similar to a traditional state machine except that it produces correctness proofs of its state transitions. Such proofs can be checked efficiently by a verifier without locally reexecuting state transitions and without access to the (plaintext) content of requests, responses, or the internal state of the machine. Consequently, VSMs enable a wide class of real-world services to prove their correct operation—without compromising privacy. For example, by appropriately programming state transitions, VSMs can implement verifiable versions of payment networks [27, 61], dark pools [63], ad exchanges [4], blockchains and smart contracts [12, 29, 48, 59], and any request-processing application that interacts with a database.

There is an elegant solution to build VSMs by employing *efficient arguments* [40, 43, 46, 47, 56, 58], a primitive that composes probabilistically checkable proofs (PCPs) [6, 7] with cryptography. Specifically, an untrusted service can maintain state (e.g., in a key-value store), run appropriate computations that manipulate that state in response to clients' requests, and produce proofs that it faithfully executed each request on the correct state. Such proofs are *succinct*, in the sense that the proofs are small (e.g., constant-sized) and are efficient to verify. In some constructions, the proofs are *zero-knowledge* [42], meaning that they reveal nothing beyond their validity: the state maintained by the service, along with the content of requests and responses, is kept private from a verifier.

While the original theory is too expensive to implement, recent systems [8, 14, 18, 25, 33, 34, 38, 49, 64, 66–68, 70, 72, 73, 75–79, 82–84] make significant progress. Beyond reducing the costs of the theory by over $10^{20}\times$, some of them can prove the correct execution of stateful computations like MapReduce jobs and database queries.

Despite this progress, the costs remain prohibitive: the service incurs several CPU-seconds per storage operation (e.g., `put`, `get` on a key-value store) when generating a proof of correct execution (§2.1, §7). This is over $10^{6}\times$ slower than an execution that does not produce proofs. Besides costs, storage primitives in prior systems support only a single writer, which limits them to a sequential model of execution. Consequently, they cannot scale out with additional resources by processing requests concurrently; this limits throughput that applications built atop prior systems can achieve.

We address these issues with *Spice*, a new system for building VSMs. Spice introduces a storage primitive with a key-value store interface, called *SetKV*, that is considerably more efficient than storage primitives used by prior systems (§3). Furthermore, SetKV admits concurrent writers with sequential consistency [52] (and in some cases linearizability [45]) semantics, and supports serializable transactions [21, 62]. This makes Spice the first system to build VSMs with support for a concurrent execution model (§4). Finally, we compose SetKV with prior and new techniques to ensure that a verifier can check the correct execution of requests using only cryptographic commitments that hide the content of requests, responses, and the state of the service (§3–5).

In more detail, SetKV extends a decades-old mechanism for verifying the correctness of memories [5, 23, 31, 35]. SetKV is based on set data structures whereas prior systems employ (Merkle) trees [25, 30] or commitments [34, 83]. This has two implications. First, the cost of a storage operation is a constant under SetKV (when amortized over a batch of operations) whereas in prior storage primitives it is logarithmic [25, 30] or linear [34, 83] in the size of the state. Second, SetKV allows concurrent writers since operations on sets—such as adding an element to a set—commute.

We implement Spice atop a prior framework [1, 78].

A programmer can express a VSM in a broad subset of C (augmented with APIs for SetKV and transactions), and compile it to executables of clients that generate requests, servers that process those requests and generate proofs, and verifiers that check the correctness of responses by verifying proofs. We build several realistic applications with Spice: an inter-bank transaction service [27], a cloud-hosted ledger [28], and a dark pool [63]. Our experimental evaluation shows that Spice's VSMs are 29–2,000× more CPU-efficient than the same VSMs built with prior work. Furthermore, they achieve 18,000–685,000× higher throughput than prior work by employing multiple CPUs. Concretely, Spice's VSMs support 488–1167 transactions/second on a cluster of 16 machines, each with 32 CPU cores and 256 GB of RAM.

Despite these advances, Spice has limitations. To achieve high throughput, Spice proves state transitions in batches, so one must wait for a batch to be verified before determining the correctness of any individual request, which introduces latency (§3, §7.2). The CPU cost to produce proofs remains large (§7.1, §7.3) when compared to an execution that does not produce proofs. Nevertheless, Spice opens the door to VSMs that support a concurrent model of computation and to many exciting applications.

2 Problem statement and background

Spice's goal is to produce *verifiable state machines (VSMs)*. We begin by reviewing state machines, which we use as an abstraction to represent a request-processing service. A state machine is specified by a tuple $(\Psi, \mathcal{S}_0)$, where Ψ is a deterministic program that encodes state transitions, and $\mathcal{S}_0$ is the initial state of the machine (e.g., a set of key-value pairs). The state machine maintains its state with $\mathcal{S}_{cur}$, which is initialized to $\mathcal{S}_0$. When the machine receives a request x, it executes Ψ with x and its state $\mathcal{S}_{cur}$ as inputs; this mutates the state of the machine and produces a response y. More formally, the machine executes a request x to produce a response y as follows:

$$(\mathcal{S}_i, y) \leftarrow \Psi(\mathcal{S}_{cur}, x)$$
$$\mathcal{S}_{cur} \leftarrow \mathcal{S}_i$$

A state machine may execute a batch of requests concurrently to achieve a higher throughput. In such a case, the *behavior* of the state machine (i.e., the state after executing a batch of requests, and the responses produced by the machine) depends on the desired correctness condition for concurrent operations. In this paper we focus on *sequential consistency* [52] as the correctness condition for concurrent operations on single objects, and *serializability* for multi-object transactions [21, 62].

A *verifiable* state machine permits the verification of state transitions without reexecution and without access to the (plaintext) contents of requests, responses, and

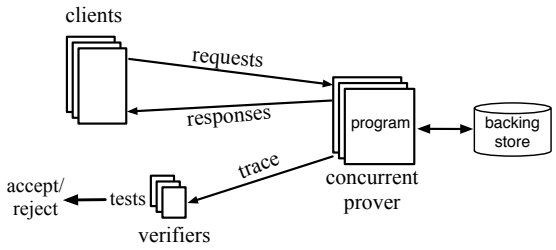


FIGURE 1—Overview of verifiable state machines (see text).

the state of the machine ($\mathcal{S}_{cur}$). Specifically, a VSM is a protocol involving a *prover* $\mathcal{P}$, a set of clients that issue requests, and one or more *verifiers* $\{\mathcal{V}_1, \ldots, \mathcal{V}_\ell\}$ that check the correctness of the execution (clients can be verifiers). We depict this protocol in Figure 1; it proceeds as follows.

1. $\mathcal{P}$ runs a state machine $(\Psi, \mathcal{S}_0)$ that processes requests concurrently and maintains its state on a persistent storage service (e.g., a key-value store).
2. Clients issue a set of requests, $x_1, \ldots, x_m$, concurrently to $\mathcal{P}$ and get back responses, $y_1, \ldots, y_m$.
3. Each verifier $\mathcal{V}_j$ receives an opaque *trace* from $\mathcal{P}$ and runs a local check on the trace that outputs accept or reject. Concretely, the trace contains a *commitment*[1] to the initial state of the machine, a commitment to the final state after executing the batch of requests, and a commitment and proof for each request-response pair.

An efficient VSM must satisfy the following properties.

- **Correctness.** If $\mathcal{P}$ is *honest* (i.e., $\mathcal{P}$'s behavior is equivalent to a correct execution of requests in a sequential order) then $\mathcal{P}$ can make a $\mathcal{V}_j$ output `true`.
- **Soundness.** If $\mathcal{P}$ errs (e.g., it does not execute Ψ or violates semantics of storage), then $\Pr[\mathcal{V}_j \text{ outputs } \texttt{true}] \leq \epsilon$, where ϵ is small (e.g., $1/2^{128}$).[2]
- **Zero-knowledge.** The trace does not reveal anything to a verifier $\mathcal{V}_j$ beyond the correctness of $\mathcal{P}$, the number of requests executed by $\mathcal{P}$, and the size of $\mathcal{P}$'s state.
- **Succinctness.** The size of each entry in the trace should be small, ideally a constant (e.g., a few hundred bytes). The cost to a $\mathcal{V}_j$ to verify an entry is linear in the size of the entry (e.g., a few milliseconds of CPU-time).
- **Throughput.** $\mathcal{P}$ should be able to execute (and generate proofs for) hundreds of requests/second.

VSMs are related to recent systems for proving the correct execution of stateful computations [8, 25, 30, 34, 38, 83]. However, in prior systems: (1) $\mathcal{P}$ lacks mechanisms

[1] A commitment c to a value x is *hiding* and *binding*. Hiding means that c does not reveal anything about x. Binding means that it is infeasible to find a value $x' \neq x$ which produces the same commitment.

[2] We discuss how to prevent $\mathcal{P}$ from *equivocating* (i.e., showing different traces to different verifiers) or omitting requests in Section 9.

to prove that it correctly executed requests concurrently, and (2) $\mathcal{P}$ incurs high CPU costs to produce proofs. Consequently, prior systems do not satisfy our throughput requirement. We provide an overview of a prior system below, but note that Spice addresses both issues.

2.1 A prior instantiation of VSMs

We now describe a prior system that implements VSMs; our goal is to introduce concepts necessary to describe Spice and to highlight why prior systems are inefficient. We focus on Pantry [25]; Section 8 discusses other work.

Programming model and API. Pantry [25] follows the VSM protocol structure introduced above. In Pantry, a state machine's program (i.e., Ψ) is expressed in a subset of C, which includes functions, structs, typedefs, preprocessor macros, if-else statements, loops (with static bounds), explicit type conversions, and standard integer and bitwise operations. For Ψ to interact with a storage service, Pantry augments the above C subset with several storage APIs; an example is the `get` and `put` API of a key-value store. Also, Pantry supports `commit` (and `decommit`) APIs to convert blobs of data (e.g., a request) into commitments (and back)—to hide data from verifiers.

Mechanics. Pantry meets the correctness, soundness, zero-knowledge, and succinctness properties of VSMs (§2). To explain how, we provide an overview of Pantry's machinery; we start with a toy computation.

```
int increment(int x) {
   int y = x + 1;
   return y;
}
```

Pantry proceeds in three steps to execute a computation.

(1) Express and compile. A programmer expresses the desired computation in the above subset of C, and uses Pantry's compiler to transform the program into a low-level mathematical model of computation called *algebraic constraints*. This is essentially a system of equations where variables can take values from a *finite field* $\mathbb{F}_p$ over a large prime p (i.e., the set $\{0, 1, \ldots, p-1\}$). For the above toy computation, Pantry's compiler produces the following system of equations (uppercase letters denote variables and lowercase letters denote concrete values):

$$\mathcal{C} = \left\{ \begin{array}{rcl} X - x & = & 0 \\ Y - (X+1) & = & 0 \\ Y - y & = & 0 \end{array} \right\}$$

A crucial property of this transformation is that the set of equations is *satisfiable*—there exists a solution (a setting of values to variables) to the system of equations—*if and only if* the output is correct. For the above constraint set, observe that if $y = x + 1$, $\{X \leftarrow x, Y \leftarrow y\}$ is a solution. If $y \neq x + 1$, then there does not exist any solution and the constraint set is not satisfiable.

(2) Solve. The prover *solves* the equations using the input x provided by the client. In other words, the prover obtains an assignment for each of the variables in the system of equations and sends the output y to the client.

(3) Argue. The prover *argues* (or proves) that the system of equations has a solution (which by the above transformation property establishes that y is the correct output of the computation with x as the input). To prove that a system of equations is satisfiable, the prover could send its solution (i.e., values for each of the variables in the equation) to a verifier, and the verifier could check that each equation is satisfiable. However, this approach meets neither the succinctness nor the zero-knowledge requirement of VSMs: the size of the proof is linear in the running time of the computation, and the solution reveals inputs, outputs, and the internal state of the computation.

To guarantee both properties, Pantry employs an argument protocol referred to as a zkSNARK [22] to encode the prover's solution to the system of equations as a short proof. Furthermore, a zkSNARK is *non-interactive* and often supports *public verifiability*, meaning that anyone (acting as a verifier) can check the correctness of proofs without having to interact with the prover. Details of how these protocols work are elsewhere [14, 18, 25, 44, 64, 78, 81]; we first focus on costs and then discuss a subset of mechanisms in Pantry that are relevant to our work.

Pantry's costs. Since costs depend on the choice of argument protocol and Pantry implements several [64, 67], we assume a recent protocol due to Groth [44]. The costs to a $\mathcal{V}_j$ are small: the proof produced by $\mathcal{P}$ and sent over the network to $\mathcal{V}_j$ per Ψ is short (128 bytes); $\mathcal{V}_j$'s cost to validate a proof is only a few milliseconds of CPU-time. $\mathcal{P}$'s costs to produce a proof scale (roughly) linearly with the number of constraints of the program; concretely, this cost is $\approx 150\mu$s of CPU-time per constraint.[3]

2.1.1 Interacting with external resources

A key limitation of the above algebraic constraint formalism is that it cannot handle interactions with the external "world" such as accessing disk, or sending and receiving packets over a network. To address this, Pantry relies on the concept of *exogenous computations*.

An exogenous computation is a remote procedure call (RPC) to an external service, which can be used to read from a disk or interact with remote servers (using OS services). Such an external service is executed outside of the constraint formalism (hence the name). The RPC simply returns a response that is then assigned to appropriate variables in the constraint set of a computation. We illustrate this concept with an example below.

[3] The time complexity and the concrete per-constraint cost we provide assume that the constraint set is produced in the *quadratic form* [40, 67]: each constraint is of the form $P_1 \cdot P_2 = P_3$, where P_1, P_2, and P_3 are degree-1 polynomials over the variables in the constraint set.

Suppose that the computation is $y = \sqrt{x}$, where x is a perfect square. Of course, one could represent the square-root function using constraints and apply the above machinery, but the resulting constraint set is highly verbose (which increases the prover's cost to solve and argue). Exogenous computations offer a way to express the equivalent (and much cheaper) computation with:

```
int sqrt(int x) {
  int y = RPC(SQRT, x); //exogenous computation
  assert(y*y == x);
  return y;
}
```

The above code compiles to the following constraint set:

$$\mathcal{C} = \left\{ \begin{array}{rcl} X - x & = & 0 \\ (Y_{exo} \cdot Y_{exo}) - X & = & 0 \\ Y_{exo} - y & = & 0 \end{array} \right\}$$

The prover computes $\sqrt{x}$ outside of constraints (e.g., by running a Python program) and assigns the result to Y_{exo} when solving the equations (Step 2). The `assert` statement becomes an additional constraint that essentially forces the prover to prove that it has verified the correctness of Y_{exo}. A similar approach can be used to interact with services like databases. The challenge is defining an appropriate `assert` statement, as we discuss next.

2.1.2 Handling state

As discussed above, exogenous computations enable a program Ψ to interact with a key-value store by issuing an RPC. This alone is insufficient because the prover is untrusted and can return any response to RPCs. For example, if the prover maintains a key-value store with the tuple (k, v), and Ψ issues an `RPC(GET, k)`; the prover could return $v' \neq v$. Consequently, as in the above `sqrt` example, Ψ must verify the result of every RPC.

To enable this verification, Pantry borrows the idea of *self-verifying* data blocks from untrusted storage systems: it names data blocks using their collision-resistant hashes (or *digests*). The following example takes as input a digest and increments the value of the corresponding data.

```
Digest increment(Digest d) {
  // prover supplies value of block named by d
  int block = RPC(GETBLOCK, d);
  assert(d == Hash(block));
  int new_block = block + 1;
  // supply to prover a new block and get digest
  Digest new_d = RPC(PUTBLOCK, new_block);
  assert(new_d == Hash(new_block));
  return new_d;
}
```

Pantry abstracts these operations with two APIs: (1) `PutBlock` which takes as input a block of data and returns its digest, and (2) `GetBlock` which returns a previously stored block of data given its digest (these APIs take care of the RPC call and the appropriate asserts and invocations of the hash function). Atop this API, Pantry builds more expressive storage abstractions using prior ideas [23, 39, 54, 57]. To support RAM, Pantry encodes the state in a Merkle tree [23, 57]. To support a key-value store, Pantry uses a *searchable* Merkle tree: an AVL tree where internal nodes store a hash of their children. To read (or update) state in these tree-based storage primitives, the program executes a series of `GetBlock` (and `PutBlock`) calls starting with the root of the tree.

Hiding requests and responses. The above storage primitive can be used to hide requests and responses from a verifier. Specifically, the prover keeps the plaintext requests and responses in its persistent storage and releases cryptographic commitments to requests and responses to a verifier. As in the `increment` example, a C program must take as input a commitment to a request, obtain the plaintext version of it using an RPC, and produce a commitment to the response. This logic is abstracted with the `commit` and `decommit` APIs.

Costs. We now assess the cost of a key-value store operation under Pantry. A `get`(k) makes $\lceil \log_2 n \rceil$ calls to `GetBlock` (where n is the number of key-value pairs), and each `GetBlock` call requires encoding a hash function as constraints (to represent the `assert` statement that verifies the return value of the RPC); a `put` requires twice as many operations. Thus, a single `get` on a key-value store that supports as few as $n = 1{,}000$ entries requires 44,000 constraints (§7.1); this translates to 6.6 CPU-seconds for producing a proof. Furthermore, in Pantry the root of a Merkle tree is a point of contention so a batch of operations cannot execute concurrently.

2.2 Outlook and roadmap

Given the overwhelming expense to execute (and produce a proof for) a simple storage operation when using a tree-based data structure, we believe that making meaningful progress requires revisiting mechanisms for verifying interactions with storage. In Section 3.1, we describe an entirely different way to verify storage operations that relies on a set—rather than a tree—data structure. In Section 3.2, we show how to employ this set-based storage primitive to realize efficient VSMs, and in Section 4 we show how, unlike Merkle trees, this set-based primitive allows requests to be processed concurrently. Finally, Section 5 describes how to instantiate the set-based storage primitive efficiently such that each `get` and `put` operation can be represented with about a thousand constraints.

3 Efficient storage operations in VSMs

This section presents a new mechanism to handle storage operations in VSMs. We first discuss the design of a *verifiable key-value store* based on set data structures; the design itself is orthogonal to VSMs and can be used to

build a stand-alone untrusted storage service. We then show to how to compose the new key-value store with prior machinery to realize efficient VSMs.

3.1 SetKV: A verifiable key-value store

The goal of a verifiable key-value store is to enable an entity $\mathcal{V}_\mathcal{K}$ to outsource a key-value store $\mathcal{K}$ to an untrusted server $\mathcal{P}_\mathcal{K}$, while being able to verify that interactions with $\mathcal{K}$ are correct. Specifically, $\mathcal{P}_\mathcal{K}$ receives operations from $\mathcal{V}_\mathcal{K}$ and executes them on $\mathcal{K}$ such that $\mathcal{V}_\mathcal{K}$ can check that a `get` on a key returns the value written by the most recent `put` to that key. This protocol proceeds as follows.

1. $\mathcal{V}_\mathcal{K}$ calls `init` to obtain an object that encodes the initial empty state of $\mathcal{K}$.
2. $\mathcal{V}_\mathcal{K}$ issues `insert`s, `get`s, and `put`s sequentially to $\mathcal{P}_\mathcal{K}$ and receives responses. $\mathcal{V}_\mathcal{K}$ locally updates its object for every request-response pair.
3. After a batch of operations, $\mathcal{V}_\mathcal{K}$ runs `audit` that computes over its local object (and auxiliary responses from $\mathcal{P}_\mathcal{K}$), and outputs whether or not $\mathcal{P}_\mathcal{K}$ operated correctly.

We desire the following properties from this protocol.

- If $\mathcal{P}_\mathcal{K}$ correctly executes operations on $\mathcal{K}$, then it can make $\mathcal{V}_\mathcal{K}$'s `audit` output `true`.
- If $\mathcal{P}_\mathcal{K}$ errs, then $\Pr\{$`audit` outputs `true`$\} < \theta$, where θ is very small (e.g., $1/2^{128}$).
- $\mathcal{V}_\mathcal{K}$ maintains little state (e.g., tens of bytes).

Figure 2 depicts our construction. We call this construction SetKV for ease of reference, but note that it introduces small—albeit critical—changes to the offline memory checking scheme of Blum et al. [23] (and its follow-up refinement [31]) and the Concerto key-value store [5]. We discuss our modifications at the end of this subsection; these changes are necessary to build VSMs using SetKV (§3.2). We prove that SetKV meets all desired properties in Appendix C.1 [65]. Below, we describe how SetKV works starting with a straw man design.

A straw man design. Suppose $\mathcal{V}_\mathcal{K}$ maintains a totally-ordered log where it records all key-value operations it issues to $\mathcal{P}_\mathcal{K}$ along with the responses supplied by $\mathcal{P}_\mathcal{K}$. $\mathcal{V}_\mathcal{K}$ can execute the following `audit` procedure: for each `get` on a key k recorded in the log, identify the most recent `put` to k (by traversing the log backwards starting from the point at which the `get` is recorded) and check if the value returned by the `get` matches the value written by the `put`. If all the checks pass, $\mathcal{V}_\mathcal{K}$ outputs `true`.

There are two issues with this straw man: (1) $\mathcal{V}_\mathcal{K}$'s log size is proportional to the number of key-value store operations and it grows indefinitely; (2) the cost to verify the correctness of each `get` is linear in the size of the log.

Mechanics of SetKV. SetKV addresses both issues associated with the straw man. It lowers verification cost by relying on two sets instead of an append-only log, and it reduces the size of the state maintained by $\mathcal{V}_\mathcal{K}$ by leveraging a particular type of cryptographic hash function that operates on sets. We elaborate on these next.

```
1:  function init( )
2:      return s ← VKState{0, 0, 0}
3:  function insert(s, k, v)
4:      ts' ← s.ts + 1
5:      RPC(INSERT, k, (v, ts'))      // P_K executes INSERT on K
6:      ws' ← s.ws ⊙ H({(k, v, ts')})
7:      return VKState{s.rs, ws', ts'}
8:  function get(s, k)
9:      (v, t) ← RPC(GET, k)          // P_K executes GET on K
10:     rs' ← s.rs ⊙ H({(k, v, t)})
11:     ts' ← max (s.ts, t) + 1
12:     RPC(PUT, k, (v, ts'))         // P_K executes PUT on K
13:     ws' ← s.ws ⊙ H({(k, v, ts')})
14:     return VKState{rs', ws', ts'}, v
15: function audit(s)
16:     rs' ← s.rs
17:     keys ← RPC(GETKEYS)  // P_K returns a list of keys in K
18:     for k in keys do
19:         (v, t) ← RPC(GET, k)      // P_K executes GET on K
20:         rs' ← rs' ⊙ H({(k, v, t)})
21:     if keys has duplicates or rs' ≠ s.ws then return false
22:     else return true
```

FIGURE 2—SetKV: A verifiable key-value store based on set data structures [5, 23, 31, 35]. The logic depicted here is run by $\mathcal{V}_\mathcal{K}$; $\mathcal{P}_\mathcal{K}$ responds to RPCs. $\mathcal{V}_\mathcal{K}$'s state consists of two set-digests and a timestamp *ts*; $\mathcal{H}$ is an incremental set collision-resistant hash function; see text for details. A `put` is similar to `get` except that lines 11 and 13 use the value being written instead of v.

(1) Using sets. Instead of a totally-ordered log, suppose that $\mathcal{V}_\mathcal{K}$ maintains a local timestamp counter *ts* along with two sets, a "read set" (*RS*) and a "write set" (*WS*). SetKV's key idea is to design a mechanism that combines all the checks in the straw man design (performed on the return value of each `get` using a log) into a single check on these two sets; if the server executes any operation incorrectly, the check fails. Of course, unlike the above log-based checks, if the set-based check fails, $\mathcal{V}_\mathcal{K}$ will not know which particular operation was executed incorrectly by $\mathcal{P}_\mathcal{K}$, but this dramatically reduces verification costs.

Details of the set-based check. First, we structure the key-value store $\mathcal{K}$ so that each entry is of the form (k, v, t) where k is a key, v is the associated value, and t is a timestamp (more precisely a Lamport clock [51]) that indicates the last time the key was read (or updated). $\mathcal{V}_\mathcal{K}$ initializes *RS* and *WS* to empty, and *ts* to 0. When $\mathcal{V}_\mathcal{K}$ wants to `insert` a new key-value pair (k, v) into $\mathcal{K}$, it increments the local timestamp *ts*, adds the tuple (k, v, ts) into *WS*, and sends this tuple to $\mathcal{P}_\mathcal{K}$. Similarly, when $\mathcal{V}_\mathcal{K}$ wishes to execute a `get` (or a `put`) operation on an

existing key k, $\mathcal{V}_\mathcal{K}$ performs the following five steps:

1. Get from $\mathcal{P}_\mathcal{K}$ via an RPC the current value v and timestamp t associated with key k
2. Add the tuple (k, v, t) into RS
3. Update the local timestamp $ts \leftarrow \max(ts, t) + 1$
4. Add the tuple (k, v', ts) into WS (where $v' = v$ for a `get`, or the new value for a `put`)
5. Send the new tuple (k, v', ts) to $\mathcal{P}_\mathcal{K}$ via an RPC

Observe that the sets maintained by $\mathcal{V}_\mathcal{K}$ preserve two important invariants: (1) every element added to RS and WS is unique because ts is incremented after each operation; and (2) RS "trails" WS by exactly the last write to each key (i.e., $RS \subseteq WS$). These lead to an efficient `audit` procedure: $\mathcal{V}_\mathcal{K}$ can request the current state of $\mathcal{K}$ (i.e., the set of key, value, and timestamp tuples) from $\mathcal{P}_\mathcal{K}$ (denote this returned set as M), and check if:

$$RS \cup M = WS$$

There is also a check in `audit` that verifies whether all the keys in M are unique. This check prevents the following double insertion attack: if $\mathcal{V}_\mathcal{K}$ issues to $\mathcal{P}_\mathcal{K}$ an `insert` operation with a key that already exists in $\mathcal{K}$, a correct $\mathcal{P}_\mathcal{K}$ should return an error message. However, a malicious $\mathcal{P}_\mathcal{K}$ could return success for both `insert`s, and in the future, return either value for a `get` on such a key.

Correctness intuition. We now use an example to provide intuition about the set-based check. Suppose that after initialization, $\mathcal{V}_\mathcal{K}$ inserts a new key-value pair (k, v) into $\mathcal{K}$ (via the above protocol). $\mathcal{V}_\mathcal{K}$'s state will be:

$$RS{=}\{\}, WS{=}\{(k, v, 1)\}, ts{=}1$$

If $\mathcal{V}_\mathcal{K}$ runs the `audit` procedure, then a correct $\mathcal{P}_\mathcal{K}$ can return its state, which in this case is simply $M = \{(k, v, 1)\}$. This leads $\mathcal{V}_\mathcal{K}$'s `audit` to return `true` since $RS \cup M{=}WS$, and the set of keys in M has no duplicates. Suppose that $\mathcal{V}_\mathcal{K}$ then calls `get`(k) and $\mathcal{P}_\mathcal{K}$ misbehaves by returning $(v', 1)$ where $v' \neq v$. $\mathcal{V}_\mathcal{K}$'s state will be updated to:

$$RS{=}\{(k, v', 1)\}, WS{=}\{(k, v, 1), (k, v', 2)\}, ts{=}2$$

Observe that for any set M, $RS \cup M \neq WS$ (this is because $RS \nsubseteq WS$). By returning an incorrect response, $\mathcal{P}_\mathcal{K}$ permanently damaged its ability to pass a future `audit`.

(2) Compressing $\mathcal{V}_\mathcal{K}$'s state. $\mathcal{V}_\mathcal{K}$ cannot track the two sets explicitly since they are larger than $\mathcal{K}$. Instead, $\mathcal{V}_\mathcal{K}$ employs a particular type of hash function $\mathcal{H}(\cdot)$ that acts on sets and produces a succinct *set-digest* [9, 31]. $\mathcal{H}$ meets two properties. First, it is *set collision-resistant*, meaning that it is computationally infeasible to find two different sets that hash to the same set-digest. Second, $\mathcal{H}$ is *incremental*: given a set-digest d_S for a set S, and a set W, one can efficiently compute a set-digest for $S \cup W$. Specifically, there is an operation $\odot$ (that takes time linear in the number of elements in W) such that:

$$\begin{aligned}\mathcal{H}(S \cup W) &= \mathcal{H}(S) \odot \mathcal{H}(W) \\ &= d_S \odot \mathcal{H}(W)\end{aligned}$$

$\mathcal{V}_\mathcal{K}$ leverages $\mathcal{H}$ to create (and incrementally update) set-digests that encode RS and WS, and it keeps these digests and the local timestamp in a small data structure:

```
struct VKState {
  SetDigest rs; // a set-digest of RS
  SetDigest ws; // a set-digest of WS
  int ts;
}
```

The same correctness argument (discussed above) applies except that we must account for the case where $\mathcal{P}_\mathcal{K}$ identifies a collision in $\mathcal{H}$, which can allow it to misbehave and still pass the `audit`. Fortunately, the probability that $\mathcal{P}_\mathcal{K}$ can find any collision is very small ($\theta \leq 2^{-128}$).

Note that while the `audit` procedure (Figure 2) appears to require $\mathcal{V}_\mathcal{K}$ to keep state linear in the size of $\mathcal{K}$ to store the set of all keys (to check for duplicates), this is not the case. If `getkeys` (Fig. 2, Line 17) returns a sorted list of keys, the uniqueness check can be expressed as a streaming computation. Consequently, $\mathcal{V}_\mathcal{K}$ only needs enough state for `VKState`, and the metadata required to track the status of the streaming computation; all of this is tens of bytes, which meets our requirement.

Differences with prior designs. SetKV supports inserting any number of keys, whereas offline memory checking protocols [23, 31, 35] have a fixed memory size. To support insertion, we add the `insert` procedure, the `getkeys` RPC, and the uniqueness check (Figure 2, Line 21). To prevent $\mathcal{P}$ from denying that a particular key has been inserted, and to disallow $\mathcal{P}$ from maintaining a key-value store with duplicate keys, we have additional checks (Appendix A.4 [65]). Concerto [5] also supports inserts but it is more expensive than SetKV since it requires $\mathcal{V}_\mathcal{K}$ to issue two additional RPCs per `insert` (and two additional calls to $\mathcal{H}$ to update rs and ws) to maintain an index of keys, so Concerto's approach is up to 3× more expensive than SetKV for $\mathcal{V}_\mathcal{K}$.

Several prior schemes [5, 23, 35] use instances of $\mathcal{H}$ that require $\mathcal{V}_\mathcal{K}$ to use cryptographic material that must be kept secret from $\mathcal{P}_\mathcal{K}$. While this is not an issue in the standalone setting presented in this section (since $\mathcal{V}_\mathcal{K}$ updates set-digests locally), it is problematic in the VSM context where the prover $\mathcal{P}$ executes these operations on behalf of clients (§3.2). In contrast, our construction of $\mathcal{H}$ does not require secret cryptographic material (§5.2). Finally, the audit procedure of SetKV does not modify $\mathcal{V}_\mathcal{K}$'s set-digests (as is the case in Concerto's), which lowers the costs of `audit` by 2×.

3.2 Building VSMs using SetKV

Spice follows an approach similar to Pantry to build VSMs. As with the Pantry baseline discussed in the prior section, Spice uses Groth's argument protocol [44] as a black box (Spice can also use many other argument protocols, as we discuss in Section 9). The principal difference between the two systems is in how they handle storage operations, which we discuss next.

Recall from Section 2.1 that a VSM's program Ψ interacts with external services (e.g., a storage service) by issuing RPCs. Since the prover is untrusted and can return incorrect responses to RPCs, Ψ must verify each RPC response via an `assert`; Section 2.1.2 discusses the verification mechanism in Pantry. We now discuss an alternate mechanism based on SetKV.

At a high level, Spice's idea is to employ SetKV's verifier (i.e., $\mathcal{V}_{\mathcal{K}}$) to check the interactions of Ψ with a storage service. To accomplish this, we build a C library that implements the `init`, `insert`, `get`, `put`, and `audit` procedures in Figure 2. A VSM programmer uses this library to write Ψ, and compiles Ψ into algebraic constraints (and client, server, verifier executables). To illustrate this idea, we start with an example in which Ψ increments an integer value associated with a key requested by a client.

```
Value increment(VKState* s, Key k) {
  Value v;
  // prover supplies value v for key k
  get(s, k, &v); //setkv library call (updates s)
  v = (Value) ((int) v + 1);
  put(s, k, v); // setkv library call (updates s)

  // batch-verify all storage operations
  assert(audit(*s) == true); // setkv library call
  return v;
}
```

Observe that the high-level structure of the above program is nearly identical to the example we discussed in the context of Pantry. A key difference, however, is that under Pantry, Ψ verifies each storage operation (e.g., `GetBlock`) with an `assert`; under Spice, Ψ verifies all storage operations at the end with a single `assert` that calls SetKV's `audit` procedure.

Costs. Since `init`, `insert`, `get`, and `put` execute a constant number of arithmetic operations (Figure 2), Spice compiles them into a constant number of equations when transforming Ψ into the constraint formalism. `audit`, however, computes over the entire state of the key-value store, so it compiles to a constraint set with size linear in the number of objects in the key-value store (say n). Fortunately, `audit` is called only once, so its costs are amortized over all storage operations in Ψ.

In more detail, if Ψ executes $O(n)$ storage operations before calling `audit`, the (amortized) cost of each storage operation is a constant. However, for the services that Spice targets (§1, §6), Ψ executes far fewer storage operations than n. This leads to an undesirable situation: the amortized cost of a storage operation can be worse than in Pantry (where each storage operation's cost is logarithmic in n). Spice addresses this by decoupling the call to `audit` from the rest of Ψ. We discuss this below.

Spice's VSMs. Let Ψ be a program with the same structure as the previous `increment` example: Ψ takes as input a request x and a `VKState` s, interacts with the storage via RPCs, verifies those interactions at the end via `assert`, updates s, and outputs a response y. Spice splits Ψ into two independent programs: Ψ_{req} and Ψ_{audit}, where Ψ_{req} is same as Ψ except that it does not have the `assert` statement at the end; Ψ_{audit} is the following program:

```
void audit_batch(VKState s) {
  assert(audit(s) == true);
}
```

This decomposition achieves the following: proving the correct execution of m instances of Ψ is *equivalent* to proving the correct execution of the corresponding m instances of Ψ_{req} and a single instance of Ψ_{audit}. By equivalent, we mean that a verifier $\mathcal{V}$ outputs `true` to $m+1$ proofs (one per instance of Ψ_{req} and Ψ_{audit}) if and only if $\mathcal{V}$ would have output `true` to the m proofs produced by instances of Ψ. Thus, if $m{=}O(n)$, the $O(n)$ constraints needed to express Ψ_{audit} are effectively amortized over the m requests, making the (amortized) number of constraints for each storage operation in Ψ_{req} a constant. Note that the costs of Ψ_{audit} can actually be amortized across *different* computations (they can be instances of different Ψ_{req}).

This approach has two drawbacks. First, it increases latency since $\mathcal{V}$ confirms the correct execution of any given instance Ψ_{req} only after it has verified all $m+1$ proofs. Second, if the proof of Ψ_{audit} fails, $\mathcal{V}$ does not learn which of the storage operations (and therefore which instance of Ψ_{req}) returned an incorrect result. However, as we show in our evaluation (§7), this decomposition reduces the cost of storage operations by orders of magnitude over Pantry, even for modest values of m.

Trace. Recall from Section 2 that each verifier $\mathcal{V}_j$ receives a *trace* from $\mathcal{P}$ to verify a batch of m instances of Ψ_{req}. This trace contains m tuples and a proof for Ψ_{audit}:

$$(x_i, s_{i-1}, y_i, s_i, \pi_i)\ \forall i \in [1,m] \quad \text{and} \quad \pi_{audit}$$

where π_i is the proof of correct execution of the i^{th} instance of Ψ_{req} with (s_{i-1}, x_i) as input and (s_i, y_i) as output. Each state s_i is an object of type `VKState` (s_0 is a `VKState` object for an empty key-value store), x_i is a request, and y_i is the corresponding response. π_{audit} establishes the correct execution of Ψ_{audit} with s_m as input.

Observe that the above trace is sufficient to guarantee correctness and soundness (since each $\mathcal{V}_j$ has all the information needed to verify the actions of $\mathcal{P}$), but it does

not satisfy zero-knowledge or succinctness. This trace is not succinct since the sizes of requests and responses could be large (they depend on the application). The trace is not zero-knowledge since requests and responses appear in plaintext. Moreover, a `VKState` object leaks the timestamp field and the set-digests (unlike commitments, hashes bind the input but do not hide it; see Footnote 1).

Commitments. To make the trace succinct and zero-knowledge, a programmer writes a VSM that takes as input (and produce as output) commitments to requests, responses, and `VKState`. For example, the programs Ψ_{req} and Ψ_{audit} discussed earlier are expressed as:

```
Commitment incr_comm(Commitment* cs, Commitment ck) {
  // prover passes value via RPC (checked by assert)
  VKState s = (VKState) decommit(*cs);
  Key k = (Key) decommit(ck);
  Value v = increment(&s, k); // prior program logic
  *cs = commit(s);
  return commit(v);
}

void audit_batch_comm(Commitment cs) {
  VKState s = (VKState) decommit(cs);
  audit_batch(s); // prior program logic
}
```

In more detail, a client sends to $\mathcal{P}$ the plaintext request x_i (k in the example). $\mathcal{P}$ computes the program (without commitments) outside of the constraint formalism and sends back to the client the output y_i (v in the example). $\mathcal{P}$ then generates a proof π_i for the version of the program that uses commitments (`incr_comm` in the example). Specifically, $\mathcal{P}$ first generates a commitment to x_i outside of the constraint formalism and uses it to solve the constraint set of Ψ_{req} (Section 9 discusses what prevents $\mathcal{P}$ from omitting requests or generating an incorrect commitment). $\mathcal{P}$ then adds to its trace commitments to each of (s_i, x_i, y_i) and the corresponding proof π_i. Each verifier $\mathcal{V}_j$ uses these commitments—instead of their plaintext versions—when verifying proofs (including π_{audit}), since the above programs use commitments as inputs and outputs. Thus, a verifier $\mathcal{V}_j$ does not learn anything about the requests, responses, or states beyond their correctness, the number of requests, and the size of the state. Also, since the size of each commitment and each proof is a constant, it satisfies the succinctness property of VSMs.

4 Supporting concurrent services

Prior instantiations of VSMs—including our design in Section 3—do not support a prover $\mathcal{P}$ that executes requests concurrently. A key challenge is producing proofs that establish that $\mathcal{P}$ met a particular consistency semantic. Note that this problem is hard even without the zero-knowledge or succinctness requirements of VSMs [71].

4.1 Executing requests concurrently

To make $\mathcal{P}$ execute requests concurrently, we introduce a concurrent version of SetKV, called *C-SetKV*, which we later integrate with Spice's design from the prior section.

C-SetKV's prover $\mathcal{P}_\mathcal{K}$ interacts with multiple instances of $\mathcal{V}_\mathcal{K}$ ($\mathcal{V}_\mathcal{K}^{(0)}, \ldots, \mathcal{V}_\mathcal{K}^{(\ell)}$) that issue `insert`, `put`, and `get` requests concurrently. C-SetKV guarantees sequential consistency [52]: an `audit` returns `true` if and only if the concurrent execution is equivalent to a sequential execution of operations and the sequential execution respects the order of operations issued by individual instances of $\mathcal{V}_\mathcal{K}$. In a few cases, C-SetKV guarantees linearizability [45]. We formalize these guarantees and provide details in Appendix C.2 [65], but the key differences between C-SetKV and SetKV are:

1. *Enforcement of isolation.* In SetKV (Figure 2), $\mathcal{V}_\mathcal{K}$ issues two RPCs for each `get` and `put`; they are executed in isolation by a correct $\mathcal{P}_\mathcal{K}$ because there is only one outstanding operation. In C-SetKV, $\mathcal{P}_\mathcal{K}$ must explicitly ensure that both RPCs are executed in isolation since it receives and executes many concurrent operations.
2. *Support for independent `VKState`s.* In SetKV, $\mathcal{V}_\mathcal{K}$ maintains a single `VKState` object that encodes its key-value store operations since initialization. In C-SetKV, each $\mathcal{V}_\mathcal{K}^{(j)}$ has its own independent `VKState` object that contains only the effects of operations issued by $\mathcal{V}_\mathcal{K}^{(j)}$.

We discuss the details of these differences below.

Enforcement of isolation. We now discuss how a correct $\mathcal{P}_\mathcal{K}$ can execute C-SetKV's four key-value store operations in isolation. It is straightforward to execute `insert` in isolation since it issues a single RPC. `audit` does not modify $\mathcal{P}_\mathcal{K}$'s state, so $\mathcal{P}_\mathcal{K}$ can executes it in isolation using a snapshot of its state. To ensure the two RPCs of `put` and `get` execute in isolation (in the presence of multiple instances of $\mathcal{V}_\mathcal{K}$), $\mathcal{P}_\mathcal{K}$ can keep track of when the first RPC starts and block any other request that attempts to operate on the same key until the second RPC (for the same key) completes. A simple approach to achieve this is for $\mathcal{P}_\mathcal{K}$ to lock a key during the first RPC and release the lock on the second RPC. A malicious $\mathcal{P}_\mathcal{K}$ could of course choose not to guarantee isolation, but as we show in Appendix C.2 [65], a future `audit` will fail. Note that in Spice, $\mathcal{P}_\mathcal{K}$ corresponds to the external storage, so the mechanism that ensures isolation happens outside of the constraint formalism (i.e., it is not encoded in Ψ).

Support for independent `VKState`s. Since each $\mathcal{V}_\mathcal{K}^{(j)}$ issues requests independently, it maintains a local `VKState` object. This creates two issues. First, the set-digests and timestamp in the `VKState` object of $\mathcal{V}_\mathcal{K}^{(j)}$ do not capture the operations issued by other instances of $\mathcal{V}_\mathcal{K}$. As a result, we need a mechanism to combine the `VKState` objects of

all instances of $\mathcal{V}_\mathcal{K}$ prior to invoking `audit`—since `audit` accepts a single `VKState` object. Second, the timestamp field *ts* is no longer unique for each operation since each $\mathcal{V}_\mathcal{K}^{(j)}$ initializes its `VKState` object with $ts = 0$. We discuss how we address these issues below.

Combining `VKState` objects. To obtain a single `VKState` object, each $\mathcal{V}_\mathcal{K}^{(j)}$ collects `VKState` objects from every other instance and locally combines all objects.[4] Combining set-digests is possible because sets are unordered and the union operation is commutative. Moreover, $\mathcal{H}(\cdot)$ preserves this property since the operation $\odot$ is commutative. As a result, each $\mathcal{V}_\mathcal{K}^{(j)}$ constructs set-digests that capture the operations of all instances of $\mathcal{V}_\mathcal{K}$ as if they were issued by a single entity. For example, the combined read set-digest is computed as $rs = rs^{(0)} \odot \ldots \odot rs^{(j)}$ (similarly for *ws*). Finally, the timestamp of the combined `VKState` object is simply 0 since it is not used in `audit`.

Handling duplicate entries. Since different $\mathcal{V}_\mathcal{K}$ instances start with the same timestamp ts=0, it is possible for two different instances to add the same element into their local set-digests (in a `VKState` object); this creates a problem when multiple `VKState` objects are combined. We use an example to illustrate the problem. Suppose there are three instances of $\mathcal{V}_\mathcal{K}$: $\mathcal{V}_\mathcal{K}^{(1)}, \mathcal{V}_\mathcal{K}^{(2)}, \mathcal{V}_\mathcal{K}^{(3)}$. Suppose $\mathcal{V}_\mathcal{K}^{(1)}$ calls `insert`(k, v), making its `VKState`:

$$ws = \mathcal{H}(\{(k, v, 1)\}), rs = \mathcal{H}(\{\}), ts = 1$$

Suppose $\mathcal{V}_\mathcal{K}^{(2)}$ and $\mathcal{V}_\mathcal{K}^{(3)}$ call `get`(k) concurrently and $\mathcal{P}_\mathcal{K}$ returns an incorrect value $v' \neq v$. Specifically, $\mathcal{P}_\mathcal{K}$ returns $(k, v', 1)$ to both, so their `VKState` object is:

$$ws = \mathcal{H}(\{(k, v', 2)\}), rs = \mathcal{H}(\{(k, v', 1)\}), ts = 2$$

Now, if each $\mathcal{V}_\mathcal{K}$ instance combines set-digests in the three `VKState` objects, they get the following (we use exponents to indicate the number of copies of an element):

$$ws = \mathcal{H}(\{(k, v, 1), (k, v', 2)^2\}), rs = \mathcal{H}(\{(k, v', 1)^2\})$$

Unfortunately, since $\mathcal{H}(\cdot)$ is a set hash function the above leads to undefined behavior: $\mathcal{H}$'s input domain is a set, but the above is a *multiset*.[5] Worse, some constructions [5] use XOR for $\odot$, so $\mathcal{H}(\{(k, v', 1)^2\} = \mathcal{H}(\{\})$ (i.e., adding an element that already exists to a set-digest removes the element!). Such a hash function would lead to the following combined set-digests:

$$ws = \mathcal{H}(\{(k, v, 1)\}), rs = \mathcal{H}(\{\})$$

For these set-digests, a $\mathcal{P}_\mathcal{K}$ can make `audit` pass by returning $M = \{(k, v, 1)\}$—even though it misbehaved by returning an incorrect value to $\mathcal{V}_\mathcal{K}^{(2)}$ and $\mathcal{V}_\mathcal{K}^{(3)}$.

[4] Exchanging `VKState` objects is easy in the context of VSMs since (commitments to) all `VKState` objects appear in the trace.

[5] A multiset is a set that can contain duplicate elements.

There are two solutions. First, we can use a $\mathcal{H}(\cdot)$ that is multiset collision-resistant (our construction in Section 5 satisfies this). In that case, even if different instances of $\mathcal{V}_\mathcal{K}$ add the same elements to their set-digests, the aggregated set-digest will track the *multiplicity* of set members (i.e., the number of times an element is added to a set-digest). If $\mathcal{P}_\mathcal{K}$ misbehaves, the aggregated *rs* will not be a submultiset of the aggregated *ws*, which prevents a future `audit` from passing (Appendix C.2 [65]). The second solution is to guarantee that there are no duplicate entries. We discuss this second solution in detail in Appendix A.1 [65].

Using C-SetKV to execute requests concurrently. $\mathcal{P}$ executes (and generates proofs for) multiple instances of Ψ_{req} simultaneously using different threads of execution (e.g., on a cluster of VMs). As before, each instance of Ψ_{req} interacts with a storage service through exogenous computation. A key difference is that unlike the design in Section 3.2, each instance of Ψ_{req} checks the response from the storage service using a different instance of C-SetKV's verifier. This is essentially the desired solution, but we now specify a few details.

A verifier $\mathcal{V}_j$ receives commitments to a set of `VKState` objects, one from each thread of execution, in $\mathcal{P}$'s trace. This means that $\mathcal{V}_j$ cannot execute the $\odot$ operator on the commitments sent by $\mathcal{P}$, since $\odot$ works on set-digests and not on commitments. To address this, $\mathcal{P}$ supports a computation Ψ_{comb} that takes as input commitments to `VKState` objects and outputs a commitment to the combined `VKState` object. That is, $\mathcal{P}$ helps $\mathcal{V}_j$ combine commitments to `VKState` objects—without revealing anything about the objects and without requiring $\mathcal{V}_j$ to trust $\mathcal{P}$ ($\mathcal{P}$ produces a proof for Ψ_{comb}). $\mathcal{V}_j$ then uses the resulting commitment in Ψ_{audit}.

4.2 Supporting transactional semantics

Many services compute over multiple key-value tuples when processing a request, so they require transactional semantics. To support such services, we first build low-level mutual-exclusion primitives. We then use these primitives to build a transactional interface to C-SetKV that guarantees serializability [21, 62]. Finally, we show how those low-level primitives can be used to build other concurrency control protocols.

Mutual-exclusion primitives. Spice supports two APIs: (1) `lock` takes as input a key and returns the current value associated with the key; and (2) `unlock` takes as input a key and an updated value, and associates the new value with the key before unlocking the key. Figure 3 depicts our implementation of these APIs by essentially decomposing SetKV's `get` and `put` (Figure 2).

In essence, these primitives provide mutual-exclusion semantics by leveraging the requirement that $\mathcal{P}_\mathcal{K}$ in C-

```
1: function lock(s, k)
2:     (v, t) ← RPC(GET, k)      // P_K executes GET and locks k
3:     rs' ← s.rs ⊙ H({(k, v, t)})
4:     ts' ← max (s.ts, t)
5:     return VKState{rs', s.ws, ts'}, v
6: function unlock(s, k, v)
7:     ts' ← s.ts + 1
8:     RPC(PUT, k, (v, ts'))     // P_K executes PUT and unlocks k
9:     ws' ← s.ws ⊙ H({(k, v, ts')})
10:    return VKState{s.rs, ws', ts'}
```

FIGURE 3—Mechanics of `lock` and `unlock` (see text).

```
1: function beg_txn(s, keys)
2:     s' ← s, vals ← [ ]
3:     for k in keys do
4:         (s', v) ← lock(s', k)
5:         vals ← vals + (v)              // append the value
6:     return s', vals
7: function end_txn(s, tuples)
8:     s' ← s
9:     for (k, v) in tuples do
10:        s' ← unlock(s', k, v)
11:    return s'
```

FIGURE 4—Mechanics of `beg_txn` and `end_txn` (see text).

SetKV must execute `GET` and `PUT` RPCs on the same key in isolation. Specifically, if a request executes `lock` on a key k, $\mathcal{P}_\mathcal{K}$ must block all operations on k until the lock-owner calls `unlock` (otherwise a future `audit` fails).

Simple transactions. We now describe how the above mutual-exclusion primitives can be used to build transactions with known read/write sets: all the keys that will be accessed are known before the transaction execution begins. Spice abstracts this transactional primitive with two APIs: (1) `beg_txn` takes as input a list of keys on which a transaction wishes to operate and returns the values associated with those keys; (2) `end_txn` takes as input the list of keys and the values that the transaction wishes to commit. Between calls to these two APIs, a program Ψ_{req} can execute arbitrary computation in Spice's subset of C.

Figure 4 depicts our implementation of these APIs. `beg_txn` calls `lock` on each key in its argument to get back the current value associated with the key. `end_txn` calls `unlock` on each key (which stores the updated value before releasing the lock). This guarantees serializability since `lock` and `unlock` ensure mutual-exclusion.[6]

General transactions. We note that a transaction executed by Ψ_{req} does not need to acquire locks on all keys involved in the transaction at once. A programmer can write a Ψ_{req} that acquires locks on keys (using `lock`) over its lifetime and then releases locks (using `unlock`). This supports transactions with arbitrary read/write sets and guarantees serializability if Ψ_{req} implements two-phase locking: all locks on keys involved in the transaction are acquired before releasing any lock. Appendix A.3 [65] discusses how to implement serializable transactions with optimistic concurrency control instead.

[6]Deadlock can be avoided by acquiring locks in a deterministic order.

5 Efficient instantiations

We now describe an efficient implementation of Ψ_{audit} and the cryptographic primitives necessary to build Spice.

5.1 Parallelizing audits

Recall from Section 3.2 that $\mathcal{P}$ periodically produces π_{audit} to prove the correct execution of Ψ_{audit}. We observe that Ψ_{audit} can be expressed as a MapReduce job; thus, $\mathcal{P}$ can use existing verifiable MapReduce frameworks [25, 34, 38] to reduce the latency of producing π_{audit} by orders of magnitude. The details (of what each mapper and reducer computes) are in Appendix A.2 [65], but we discuss the costs. This approach increases each verifier's CPU costs and the size of π_{audit} by a factor of $|\text{mappers}| + |\text{reducers}|$. This is because each mapper and reducer generates a separate proof.[7] This is an excellent trade-off since checking π_{audit} is relatively cheap: 3 ms of CPU-time to check a mapper's (or a reducer's) proof, and each proof is 128 bytes.

5.2 Efficient cryptographic primitives

Set hash function. Recall from Section 3.2 that Spice represents the logic of SetKV's $\mathcal{V}_\mathcal{K}$ (Figure 2) in constraints. An important component is encoding $\mathcal{H}(\cdot)$ as a set of equations; all other operations in $\mathcal{V}_\mathcal{K}$ (such as comparisons and integer arithmetic) are already supported by the existing framework (§6). Spice instantiates $\mathcal{H}(\cdot)$ using MSet-Mu-Hash [31] defined over an elliptic curve EC:

$$\mathcal{H}(\{e_1, \ldots e_\ell\}) = \sum_{i=1}^{\ell} H(\{e_i\})$$

where $H(\cdot)$ is a *random oracle* that maps a multiset of elements to a point in EC, and point addition is the group operation. We use an elliptic curve group since prior work [17, 34, 50] shows how to express elliptic curve operations with only a handful of constraints.

However, one issue remains: we need a candidate for $H(\cdot)$ with an efficient representation as a constraints set. Our starting point for $H(\cdot)$ is $H(\cdot) = \phi(R(\cdot))$, where $R(\cdot)$ is a random oracle (instantiated using a collision-resistant hash function). R takes as input a multiset of elements and outputs an element of a set S (e.g., SHA-256 maps

[7]CTV [30] avoids the cost increase for a verifier, but incurs >10× higher expense for $\mathcal{P}$. The recent work of Wu et al. [82] offers an alternative by distributing $\mathcal{P}$'s work for any computation in a blackbox manner; applying it to `audit_batch` is future work.

an arbitrary length binary string to a 256-bit string); $\phi(\cdot)$ maps elements in S uniformly to a point in EC.

A challenge is that building $\phi(\cdot)$ using prior techniques [36] is expensive; more critically, common hash functions (e.g., SHA-256, Keccak) perform bitwise operations (XOR, shift, etc.), which are expensive to express with algebraic constraints (it takes at least 1 constraint for each bit of the inputs) [64, 69]. We discuss our solution in detail in Appendix B [65], but we make the following contribution. We show that the requirement that $H(\cdot)$ be a random oracle can be relaxed (we still require its constituent $R(\cdot)$ to be a random oracle). We leverage this relaxation to construct an efficient $\phi(\cdot)$ from Elligator-2 [20]; to build $R(\cdot)$, we use a relatively new block cipher called MiMC [2], which is more efficient than SHA-256 in the constraints formalism. In summary, our construction of $\mathcal{H}(\cdot)$ requires 10,000× fewer constraints than using SHA-256 and a prior construction for $\phi(\cdot)$ [36].

Commitments. Pantry [25] employs HMAC-SHA256 to implement `commit()` but requires ≈ 250,000 constraints to generate a commitment to a 150-byte message. Spice takes a different approach. For a message $x \in \mathbb{F}_p$ (recall from §2.1 that constraint variables are elements in $\mathbb{F}_p$), a commitment is $(x+t, R(t))$ where $t \in \mathbb{F}_p$ is a randomly-chosen value and $R(\cdot)$ is the MiMC-based random oracle introduced above. This is binding because $R(t)$ binds t due to the collision-resistance of $R(\cdot)$. It is hiding because $x+t$ is uniformly random; hence the tuple $(x+t, R(t))$ is independent of the message x. Finally, the scheme generalizes to larger messages $x \in \mathbb{F}_p^k$ in two ways: commit to each component of x independently (which increases the size of the commitment by k times), or output $(R(x)+t, R(t))$. Compared to Pantry's HMAC-SHA256, Spice's commitments require ≈300× fewer constraints.

6 Implementation and applications

We build Spice atop `pequin` [1], which provides a compiler to convert a broad subset of C to constraints, and links to `libsnark` [55] for the argument protocol (step 3; §2.1). We extend this compiler with Spice's SetKV API (including transactions and commitments) based on the design discussed in Sections 3–5. Spice uses `leveldb` [41] as its backing store to provide persistent state. In total, Spice adds about 2,000 LOC to Pequin. Our implementation of the applications discussed below consists of 1,300 lines of C and calls to Spice's API.

6.1 Applications of Spice

We built three applications atop Spice. These applications require strong integrity and privacy guarantees, and have transactions on state that can be executed concurrently. Furthermore, they tolerate batch verification (i.e., $\mathcal{P}$ can produce π_{audit} after many requests) since clients can levy financial penalties if they detect misbehavior ex post facto.

```
// pk_c is the public key of the caller
issue(VKState* s, PK pk_c, PK pk, Asset as, int a) {
  return insert(s, pk||as, a); // || is concatenation
}

retire(VKState* s, PK pk, Asset as, int a) {
  Value v[1];
  beg_txn(s, [pk||as], v); // updates s and v
  if (v[0] >= a) v[0] -= a;
  end_txn(s, [(pk||as, v[0])]); // updates s
}

// pk1, pk2 are the keys of caller and recipient
transfer(VKState* s, PK pk1, PK pk2, Asset as, int a) {
  Value v[2];
  beg_txn(s, [pk1||as, pk2||as], v); // updates s, v
  if (v[0] >= a) { v[0] -= a; v[1] += a; }
  end_txn(s, [(pk1||as, v[0]), (pk2||as, v[1])]);
}
```

FIGURE 5—Pseudocode for a Sequence-like app using Spice's API. The requests, except the public key of the caller, are wrapped in commitments; however, this part is not depicted.

Cloud-based ledger service. We consider a cloud-hosted service that maintains a ledger with balances of assets for different clients. Examples of assets include currency in a mobile wallet (e.g., Square, WeChat) and credits in a ride-sharing application. Clients submit three types of requests: `transfer`, `issue`, and `retire`. `transfer` moves an assert from one client to another, whereas `issue` and `retire` move external assets in and out of the ledger. For example, in WeChat, clients move currency from their bank accounts to their mobile wallets. This application is inspired by Sequence [28]. However, to verify the correct operation of Sequence, a verifier needs access to sensitive details of clients' requests (e.g., the amount of money) and the service's state. We address this limitation by implementing a Sequence-like service as a VSM using Spice. The ledger maintained by the service is the VSM's state and the request types discussed above are state transitions. Figure 5 depicts our implementation of this application in Spice's programming model.

Payment networks. Our second application is a payment network inspired by Solidus [27]. Banks maintain customer balances, and customers submit requests to move money from their accounts to other accounts (in the same bank or a different bank). This is similar to the previous application except that it also supports an inter-bank transfer. For such a transfer, the sender and recipient's banks must coordinate out-of-band: the sender's bank executes the `debit` part of a transfer and the recipient's bank executes the `credit` part. A verifier can check that banks are processing requests correctly without learning the content of requests: destination account, amount, etc.

A securities exchange (dark pool). A securities exchange is a service that allows buyers to bid for securities

(e.g., stock) sold by sellers. The service maintains an *order book*—a list of buy and sell orders sorted by price. Clients `submit` buy or sell orders to the service, who either fulfills the order if there is a match, or adds the order to the order book. Although traditional exchanges are public (clients can see the order book), private exchanges (or *dark pools*) have gained popularity in light of attacks such as "front-running" [63]. Dark pools, however, are opaque; indeed, there are prior incidents where dark pools have failed to match orders correctly [37, 60].

We implement the exchange as a VSM: the order book is the state, and *submit* and *withdraw* order are state transitions. At a high level, we represent the sorted order book as a doubly-linked list using Spice's storage API. Then, `submit` removes or inserts nodes to the list depending on whether there is a match or not, and `withdraw` removes nodes from the list. With Spice, verifiers learn nothing about the orders beyond the identity of the submitter, and yet they can check the correct operation of the exchange.

7 Experimental evaluation

We answer the following questions in the context of our prototype implementation and applications (§6).

1. How does Spice compare to prior work?
2. How well does Spice scale with more CPUs?
3. What is the performance of apps built with Spice?

Baselines. We compare Spice to two prior systems for building VSMs: Pantry [25] and Geppetto [34]. Sections 2.1 and 8 provide details of their storage primitives, but briefly, Pantry's storage operations incur costs logarithmic in the size of the state (due its use of Merkle trees), and the costs are linear in the size of the state in Geppetto. Besides these baselines, we consider a Pantry variant, which we call *Pantry+Jubjub*, that uses a Merkle tree instantiated with a recent hash function [32]. Finally, we compare our payment network app (§7.3) to Solidus [27].

Setup and metrics. We use a cluster of Azure D64s_v3 instances (32 physical CPUs, 2.4 GHz Intel Xeon E5-2673 v3, 256 GB RAM) running Ubuntu 17.04. We measure CPU-time, storage costs, and network transfers at the prover $\mathcal{P}$ and each verifier $\mathcal{V}_j$, and the throughput and latency of $\mathcal{P}$. Finally, we measure Spice's performance experimentally, but estimate baselines' performance through microbenchmarks and prior cost models; we use the same argument protocol for Spice and the baselines, so $\mathcal{P}$'s CPU costs in all the systems scale (roughly) linearly with the number of constraints of a Ψ.

Microbenchmarks. To put our end-to-end results in context, we measure the costs to each $\mathcal{V}_j$ and $\mathcal{P}$ in Spice's underlying argument protocol (§6), and the number of constraints needed to represent Spice's cryptographic primitives. Figure 6 depicts our results.

costs of argument protocol (§2.1, §6)	
$\mathcal{P}$'s CPU-time per constraint	$\approx 149\,\mu$s
$\mathcal{V}$'s CPU-time to check a proof	≈ 3 ms
size of a proof	128 bytes
#constraints for basic primitives (§5.2)	
random oracle $R(\cdot)$ on a 32-byte message	167
map $\phi(\cdot)$ on a 32-byte element to *EC*	105
add two points in *EC* (i.e., $\odot$ in §3.1)	8
commit to a 32-byte message	168

FIGURE 6—Microbenchmarks.

7.1 Spice's approach to state VS. prior solutions

We consider a computation Ψ that invokes a batch of `get` (or `put`) operations on a key-value store preloaded with a varying number of key-value pairs; each key and each value is 64 bits. Our metric here is the number of constraints required to represent a storage operation. Figure 7 depicts the cost of different key-value store operations under Spice and our baselines. For Spice, the reported costs include error-checking code that prevents $\mathcal{P}$ from claiming that a key does not exist (Appendix A.4 [65]).

We find that the cost of a storage operation is lower for Spice than prior works as long as $\mathcal{P}$'s state contains at least a few hundred key-value pairs. As an example, for a `get` on 1M key-value pairs in $\mathcal{P}$'s state, Spice requires 57× fewer constraints than Pantry, 29× fewer than Pantry+Jubjub, and 2,000× fewer than Geppetto.

However, Spice must execute (and produce a proof for) Ψ_{audit}, which requires constraints linear in the size of the state (§3.2). Fortunately, this can be amortized over a batch of m operations on state. Naturally, if $m = 1$ (i.e., we run Ψ_{audit} after every storage operation), then Spice's costs are higher than prior systems. But even for modest values of m, Spice comes out on top. For example, when the state is 1M key-value pairs, $m \geq 6{,}920$ is sufficient to achieve per-operation costs that are lower than Pantry. Furthermore, each request in our applications (e.g., financial transactions) perform multiple storage operations; the number of requests per batch that must be verified to outperform the baselines is much smaller.

7.2 Benefits of Spice's concurrent execution

We now assess how well Spice's prover $\mathcal{P}$ can leverage multiple CPUs and concurrent execution to achieve better throughput. For these experiments, we assume $\mathcal{P}$ executes Ψ_{audit} periodically in the background (e.g., every minute). We discuss Spice's throughput, latency, and the amortized costs of operations as a function of audit frequency.

Throughput. We setup $\mathcal{P}$ with a key-value store preloaded with 1M key-value pairs. We then have $\mathcal{P}$ run Ψ_{req} instances on a varying number of CPU cores, where each instance invokes a batch of `get` (or `put`) operations; Ψ_{req} selects keys according to two different distributions:

	get cost			put cost		
size of state (# key-value pairs)	1	10^3	10^6	1	10^3	10^6
Pantry	4.1K	44.9K	85.7K	8.2K	89.8K	171.5K
Geppetto	3	3.0K	3.0M	4	4.0K	4.0M
Pantry+Jubjub	2.1K	23.1K	44.1K	4.2K	46.2K	88.2K
Spice	1.5K	1.5K	1.5K	1.5K	1.5K	1.5K
Ψ_{audit}	1250/m	561K/m	582M/m	561/m	561K/m	582M/m

FIGURE 7—Per-operation cost of `get` and `put`—in terms of number of algebraic constraints—for Spice and its baselines with varying number of key-value pairs in $\mathcal{P}$'s state. We also depict the costs for Spice's Ψ_{audit}; m denotes the number of storage operations after which $\mathcal{P}$ runs Ψ_{audit} to produce π_{audit}. Figure 6 depicts $\mathcal{P}$'s and each $\mathcal{V}_j$'s CPU-time as a function of the number of constraints.

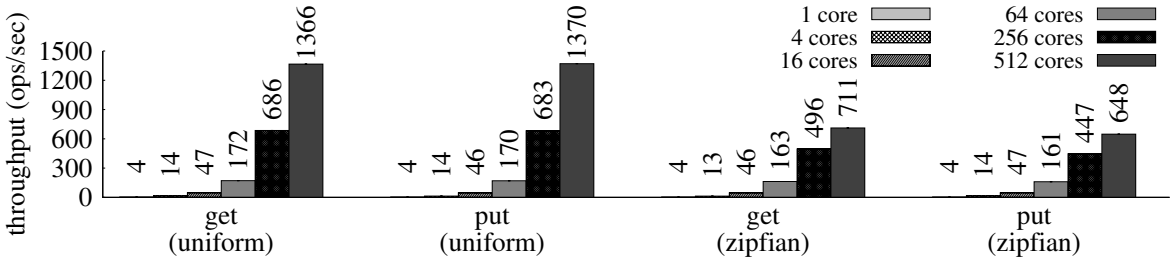


FIGURE 8—Benefits of Spice's concurrent request execution. The workload is a stream of `get`s or `put`s and $\mathcal{P}$'s state contains 1M key-value pairs. The keys are chosen uniformly at random or follow a Zipfian distribution (exponent of 1.0).

	get	put
Pantry	0.078	0.039
Pantry+Jubjub	0.153	0.076
Geppetto	0.002	0.002
Spice (1-thread)	3.6	3.6
Spice (512-threads)	1366	1370

FIGURE 9—Throughput (ops/sec) for `get` and `put` in Spice and its baselines. The size of the state is 1M key-value pairs.

uniform and Zipfian (exponent of 1.0). We measure the number of storage operations performed (and proofs produced) by $\mathcal{P}$ per second. Figure 8 depicts our results.

We find that Spice's prover achieves a near-linear speedup with increasing number of cores. When keys are chosen uniformly, $\mathcal{P}$ (with 512 cores) achieves 379× higher throughput compared to a single-core execution (for both `get` and `put` workloads). When the workload is Zipfian, the speedup is 180× due to higher contention (recall from Section 4.1 that $\mathcal{P}$ locks keys outside of the constraint formalism to guarantee isolation). In absolute terms, Spice's prover executes 648–1,370 key-value store operations/second on 512 CPU cores.

Compared to its baselines (Figure 9), Spice's throughput is 92× that of Pantry, 47× that of Pantry+Jubjub, and 1,800× that of Geppetto for `put`s. The gap widens when Spice leverages 512 cores: Spice' throughput is 35,100× higher than Pantry, 18,000× higher than Pantry+Jubjub, and 685,000× higher than Geppetto.

Latency. $\mathcal{P}$ needs additional resources to periodically produce π_{audit}. Meanwhile, the time that $\mathcal{P}$ needs to generate π_{audit} dictates the latency of storage operations—since a verifier $\mathcal{V}_j$ must check π_{audit} before establishing the correctness of prior storage operations (§3.2). We start by measuring $\mathcal{P}$'s time to run Ψ_{audit} and produce π_{audit}.

Recall from Section 5.1 that the cost of generating π_{audit} scales linearly with the size of $\mathcal{P}$'s state and we parallelize this using MapReduce (§5.1). We experiment with $\mathcal{P}$'s state containing 1M key-value pairs. We run a MapReduce job on 1,024 CPU cores consisting of 1,024 mappers, where each mapper reads 1,024 key-value tuples and produces a single set-digest (the details of the MapReduce job are in Appendix A.2 [65]). We then run 33 reducers (split over two levels containing 32 and 1 reducers) and a final aggregator. We find that the job (including proof generation) takes 3.63 minutes. As a result, if $\mathcal{P}$ runs Ψ_{audit} every k minutes the latency of any key-value store operation is at most $k + 3.63$ minutes.

Amortized costs of storage operations. Suppose we set k=10 minutes, which covers a batch of 800,000 storage operations (recall that $\mathcal{P}$ executes 1,360 ops/sec under a uniform distribution). The amortized cost of Ψ_{audit} would be $582 \cdot 10^6/800,000 \approx 728$ constraints, and the per-operation storage cost (in terms of #constraints) would be 728 + 1500 ≈ 2228 constraints. This is 76× lower than Pantry, 39× lower than Pantry+Jubjub, and 1790× lower than Geppetto for `put` operations (1M key-value pairs in $\mathcal{P}$'s state). With larger k (larger latency), this gap widens.

Verifier's costs. A verifier's costs to check a proof of correct execution for a Ψ_{req} is 3 ms of CPU-time; the

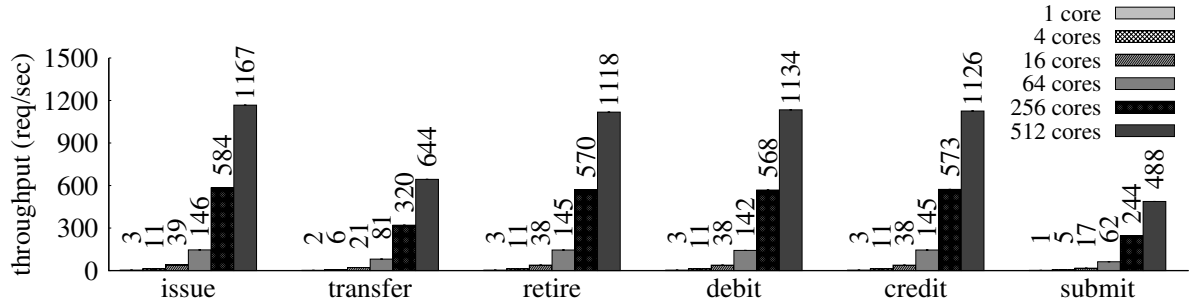


FIGURE 10—Throughput (requests processed/second) for the various applications (§6). Requests of type issue, transfer, and retire are for the cloud-based ledger service (Figure 5); issue, transfer, retire, debit, and credit are for the payment network application; and, submit requests are for the dark pool application.

proof itself is only 128 bytes (Figure 6). As we discuss in Section 5.1, the size of a proof and cost to verify Ψ_{audit} depends on the chosen MapReduce parameters. In particular, the size of π_{audit} is $(M+R+1)\cdot 128$ bytes since each mapper and each reducer produce a different proof, and verifying the entire proof takes $(M+R+1)\cdot$ 3 ms. For the above MapReduce job (M=1024, R=33), checking π_{audit} takes 3.2 CPU-seconds.

7.3 Performance of apps built with Spice

We now assess whether Spice's prover $\mathcal{P}$ meets our throughput requirement (§2). We experiment with the applications that we built using Spice (§6). Specifically, we run a concurrent $\mathcal{P}$ with a varying number of CPUs and measure its throughput for different transaction types (e.g,. `credit`, `debit`). The keys for various requests are chosen according to both uniform and Zipfian distributions, and requests compute over a million key-value pairs.

Figure 10 depicts our results for the uniform distribution case; for the Zipfian case, the throughput is 2–3.3× lower due to higher contention. Across the board, $\mathcal{P}$ achieves a near-linear speedup in transaction-processing throughput with a varying number of CPUs. Furthermore, when using 512 CPU cores, $\mathcal{P}$ achieves 488–1167 requests/second, which exceeds our throughput requirement. We now discuss the specifics of each application.

Cloud-based ledger service. Among the three transaction types supported by our first application, `issue` and `retire` involve a single storage operation whereas `transfer` requires two (to update the balances at the sender and the recipient of a transaction). Note that these storage operations are in addition to various checks on balances (see Figure 5). However, in terms of the number of constraints, storage operations dominate. As a result, $\mathcal{P}$'s throughput for `issue` and `retire` is about 2× higher than that of `transfer`. Furthermore, the throughput for `issue` and `retire` is roughly the throughput that Spice's prover achieves for a `get` (or a `put`) workload (Figure 8).

Payment networks. We only experiment with inter-bank transaction types: `credit` and `debit` (intra-bank `transfers` are the same as in our first application). These transactions involve one storage operation, so $\mathcal{P}$'s throughput is similar to `issue` and `retire` in the first application. We compare with Solidus [27], which achieves similar guarantees as our app with specialized machinery. Solidus with 32K accounts (i.e., key-value tuples) achieves 20 storage ops/sec and up to 10 tx/sec, whereas Spice's payment network on 512 CPU cores supports >1,000 tx/sec (100× higher throughput). Note that unlike our implementation, Solidus hides the sender's identity in a transaction from a verifier; achieving this in our context is future work.

Dark pools. Our third app supports two transactions, `submit` and `withdraw`. We depict only `submit` because `withdraw` has similar costs. $\mathcal{P}$ achieves 488 tx/second. This is lower than our other apps because the dark pool application is more complex: the state is a linked list layered on top of a key-value store (where each operation on the linked list is multiple storage operations), and transactions manipulate the linked list to process orders (§6.1).

8 Related work

Proving correct executions via efficient arguments. The problem of proving the correct execution of a computation is decades old [7]; many systems have reduced the expense of this theory (see [81] for a survey of this progress). While early works [33, 49, 64, 66, 68, 70, 72, 73, 75] support only stateless computations, recent systems [8, 14, 18, 25, 30, 34, 38, 78, 83, 84] support state. Section 2.1 discusses the approach in Pantry [25]; below, we discuss other approaches and how they relate to Spice.

Ben-Sasson et al. [14, 18], Buffet [78], and vRAM [84] propose a RAM abstraction based on permutation networks [13, 19, 80]. This technique can be more efficient than using Merkle trees. For example, Buffet [78] shows that each RAM operation (load, store, etc.) can be represented with several hundred constraints (compared to tens of thousands under Pantry's RAM). However, the permutation networks technique cannot be used to maintain state that persists across different request executions—a requirement of VSMs (§2).

Geppetto [34] can transfer values associated with program variables (`int`, `char`, etc.) from one computation to another. To support this, Geppetto introduces custom machinery that requires a single constraint per value transferred, so this is more efficient than Pantry for certain scenarios (e.g., sending output of a mapper as input to a reducer in MapReduce). However, it is not a good substitute to Merkle trees for key-value stores (or RAM): each storage operation requires scanning all the state. Fiore et al. [38] hybridize Geppetto-style and Pantry-style storage primitives, but it incurs the same costs as Pantry to support a key-value store.

ADSNARK [8] supports computations over state represented with an authenticated digest, but this approach does not support transferring state to other computations. vSQL [83] builds a storage primitive by representing state (e.g., a database table) as a polynomial. However, this storage primitive has the same issue as Geppetto: reading or updating a single value of the state (e.g., a row) inside a Ψ_{req} requires scanning the entire state.

Compared to prior systems, Spice proposes a cheaper and more expressive storage primitive (under a batch verification setting): Spice supports a transactional key-value store (§3, §4), which makes it possible to build useful services with plausible performance (§6–§7). Two exceptions: (1) for random access over state within a single computation, permutation networks are more efficient (indeed, Spice relies on Buffet for RAM within threads); (2) for intermediate state in a MapReduce job, Geppetto-style state transfer can be more efficient.

Concurrent systems with verifiability. Spice's use of offline memory checking [23, 31] is inspired by Concerto [5], but there are three differences. First, Concerto is limited to a key-value store whereas Spice supports (arbitrary) concurrent services expressed in a large subset of C. Second, Spice supports transactional semantics whereas Concerto is limited to single-object key-value operations. Finally, Concerto requires trusted hardware (e.g., Intel SGX) to run $\mathcal{V}_{\mathcal{K}}$. It is possible to avoid trusted hardware by letting clients act as verifiers, but the resulting system would expose the content of the key-value store (along with requests and responses); it would not guarantee zero-knowledge or succinctness (§2).

Orochi [71] enables verifiability for concurrent applications (and the underlying data store) running on an untrusted server. Orochi's key technique is a clever reexecution of all requests at the verifier—one that accommodates concurrent execution of requests at the server. Compared to Spice, Orochi imposes minimal overheads to the server. However, Orochi's verifier must keep a full copy of the server's state to verify requests along with contents of all requests and the corresponding responses. Consequently, Orochi does not satisfy the zero-knowledge or succinctness properties of VSMs (§2).

9 Discussion and summary

Equivocation and omission. Spice's $\mathcal{P}$ proves its correct operation by producing a trace that is checked by verifiers. However, $\mathcal{P}$ can equivocate: it can expose different traces to different verifiers. If the set of verifiers form a *permissioned* group (i.e., admitting new verifiers requires approval from a quorum of existing verifiers), then verifiers can agree on a single trace by employing traditional distributed consensus [26, 53], thus preventing equivocation. If the set of verifiers is unbounded, $\mathcal{P}$ can embed metadata about its trace in a permissionless blockchain [74]. Besides equivocation, $\mathcal{P}$ can omit clients' requests. To address this, clients must check if their requests are included in the trace agreed upon by verifiers.

Fault-tolerance. We can make Spice's services fault-tolerant via standard techniques. This does not require implementing a replication protocol as a VSM. This is because Spice's services maintains their internal state in a database (Spice uses leveldb), and interacts with it via RPCs (§2.1). Thus, the service could instead keep the state in a fault-tolerant storage system (e.g., DynamoDB).

Trusted setup. Spice can use many different argument protocols, but our implementation employs an argument [44] that requires a *trusted setup*: a trusted party must create cryptographic material that depends on Ψ but not on inputs or outputs to Ψ. In our context (§6), such a trusted setup can be executed by a verifier (if there is a single verifier), or in a distributed protocol [15] (when there is more than one verifier). Recent arguments [3, 10, 11, 16, 24, 79] do not require such a trusted setup. We leave it to future work to integrate them with Spice and explore trade-offs.

Summary. Spice is a substantial improvement over prior systems that implement VSMs: it improves transaction-processing throughput by over four orders of magnitude. And, although Spice's absolute costs (e.g., prover's CPU-time) are large, it enables a new set of realistic services by opening up a concurrent model of computation and achieving throughputs of over a thousand transactions/second.

Acknowledgments

We thank Weidong Cui, Esha Ghosh, Jay Lorch, Ioanna Tzialla, Riad Wahby, Michael Walfish, the OSDI reviewers, and our shepherd, Raluca Ada Popa, for helpful comments that significantly improved the content and presentation of this work. We also thank Ben Braun for help with enhancing `pequin`. We benefited from insightful conversations with Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy about Concerto [5], and Melissa Chase and Michael Naehrig about multiset hash functions. Sebastian Angel was partially funded by AFOSR grant FA9550-15-1-0302, and NSF CNS-1514422.

References

[1] Pequin: An end-to-end toolchain for verifiable computation, SNARKs, and probabilistic proofs. `https://github.com/pepper-project/pequin`, 2016.

[2] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2016.

[3] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[4] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference*, 2013.

[5] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3), May 1998.

[7] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1991.

[8] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[9] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1997.

[10] E. Ben-Sasson, I. Ben-Tov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, and M. Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2017.

[11] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[12] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[13] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2013.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2013.

[15] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[16] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. Cryptology ePrint Archive, Report 2018/828, 2018.

[17] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2014.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the USENIX Security Symposium*, 2014.

[19] V. Beneš. *Mathematical theory of connecting networks and telephone traffic*. Mathematics in Science and Engineering. Elsevier Science, 1965.

[20] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[21] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[22] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[23] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[24] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[25] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[26] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[27] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi. Solidus: Confidential distributed ledger transactions via PVORM. In *Proceedings of the ACM*

Conference on Computer and Communications Security (CCS), 2017.

[28] Chain. Introducing Sequence. https://blog.chain.com/introducing-sequence-e14ff70b730, 2017.

[29] J. P. M. Chase. ZSL Proof of Concept. https://github.com/jpmorganchase/quorum/wiki/ZSL, 2017.

[30] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2015.

[31] D. Clarke, S. Devadas, M. V. Dijk, B. Gassend, G. Edward, and S. Mit. Incremental multiset hash functions and their application to memory integrity checking. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2003.

[32] Z. E. C. Company. What is Jubjub? https://z.cash/technology/jubjub.html, 2017.

[33] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[34] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[35] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography Conference (TCC)*, 2009.

[36] R. R. Farashahi, P. Fouque, I. E. Shparlinski, M. Tibouchi, and J. F. Voloch. Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *Mathematics of Computation*, 82(281), 2013.

[37] Financial Industry Regulatory Authority. FINRA fines Goldman Sachs Execution & Clearing, L.P. $800,000 for failing to prevent trade-throughs in its alternative trading system. https://www.finra.org/newsroom/2014/finra-fines-goldman-sachs-execution-clearing-lp-800000-failing-prevent-trade-throughs, July 2014.

[38] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[39] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[40] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[41] S. Ghemawat and J. Dean. LevelDB: a fast and lightweight key/value database library by Google. https://github.com/google/leveldb, 2011.

[42] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1985.

[43] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2010.

[44] J. Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[45] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[46] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity*, 2007.

[47] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1992.

[48] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[49] A. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *Proceedings of the USENIX Security Symposium*, 2014.

[50] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. C∅C∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015.

[51] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, July 1978.

[52] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[53] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[54] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[55] libsnark. A C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark, 2012.

[56] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zeroknowledge arguments. In *Theory of Cryptography Conference (TCC)*, 2012.

[57] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1988.

[58] S. Micali. CS proofs. In *Proceedings of the IEEE*

Symposium on Foundations of Computer Science (FOCS), 1994.

[59] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[60] L. Moyer. Regulators aren't done with 'dark pool' investigations. `https://www.nytimes.com/2016/02/02/business/dealbook/regulators-arent-done-with-dark-pool-investigations.html`, Feb. 2016.

[61] N. Narula, W. Vasquez, and M. Virza. zkLedger: Privacy-preserving auditing for distributed ledgers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[62] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4), Oct. 1979.

[63] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Conference on Auctions, Market Mechanisms and Their Applications*, 2015.

[64] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[65] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge (extended version). Cryptology ePrint Archive, Report 2018/907, Sept. 2018.

[66] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[67] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[68] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[69] S. Setty, V. Vu, N. Panpalia, B. Braun, M. Ali, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). Cryptology ePrint Archive, Report 2012/598, 2012.

[70] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[71] C. Tan, L. Yu, J. B. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the Web. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[72] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[73] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[74] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin, 2017.

[75] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[76] R. S. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish. Verifiable ASICs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[77] R. S. Wahby, Y. Ji, A. J. Blumberg, abhi shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[78] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[79] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[80] A. Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, Jan. 1968.

[81] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: From theoretical possibility to near practicality. *Communications of the ACM*, 58(2), Jan. 2015.

[82] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. DIZK: A distributed zero-knowledge proof system. In *Proceedings of the USENIX Security Symposium*, 2018.

[83] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[84] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

The FuzzyLog: A Partially Ordered Shared Log

Joshua Lockerman
Yale University

†Jose M. Faleiro
UC Berkeley

†Juno Kim
UC San Diego

†Soham Sankaran
Cornell University

Daniel J. Abadi
University of Maryland, College Park

James Aspnes
Yale University

Siddhartha Sen
Microsoft Research

†Mahesh Balakrishnan
Yale University / Facebook

† *Work done while authors were at Yale University*

Abstract

The FuzzyLog is a partially ordered shared log abstraction. Distributed applications can concurrently append to the partial order and play it back. FuzzyLog applications obtain the benefits of an underlying shared log – extracting strong consistency, durability, and failure atomicity in simple ways – without suffering from its drawbacks. By exposing a partial order, the FuzzyLog enables three key capabilities for applications: linear scaling for throughput and capacity (without sacrificing atomicity), weaker consistency guarantees, and tolerance to network partitions. We present Dapple, a distributed implementation of the FuzzyLog abstraction that stores the partial order compactly and supports efficient appends / playback via a new ordering protocol. We implement several data structures and applications over the FuzzyLog, including several map variants as well as a ZooKeeper implementation. Our evaluation shows that these applications are compact, fast, and flexible: they retain the simplicity (100s of lines of code) and strong semantics (durability and failure atomicity) of a shared log design while exploiting the partial order of the FuzzyLog for linear scalability, flexible consistency guarantees (e.g., causal+ consistency), and network partition tolerance. On a 6-node Dapple deployment, our FuzzyLog-based ZooKeeper supports 3M/sec single-key writes, and 150K/sec atomic cross-shard renames.

1 Introduction

Large-scale data center systems rely on control plane services such as filesystem namenodes, SDN controllers, coordination services, and schedulers. Such services are often initially built as single-server systems that store state in local in-memory data structures. Properties such as durability, high availability, and scalability are retrofitted by distributing service state across machines. Distributing state for such services can be difficult; their requirement for low latency and high responsiveness precludes the use of external storage services with fixed APIs such as key-value stores, while custom solutions can require melding application code with a medley of distributed protocols such as Paxos [29] and Two-Phase Commit (2PC) [21], which are individually complex, slow/inefficient when layered, and difficult to merge [40, 60].

A recently proposed class of designs centers on the *shared log abstraction*, funneling all updates through a globally shared log to enable fault-tolerant databases [9–11, 19, 51], metadata and coordination services [8, 12], key-value and object stores [3, 41, 57], and filesystem namespaces [50, 56]. Services built over a shared log are simple, compact layers that map a high-level API to append/read operations on the shared log, which acts as the source of strong consistency, durability, failure atomicity, and transactional isolation. For example, a shared log version of ZooKeeper uses 1K lines of code, an order of magnitude lower than the original system [8].

Unfortunately, the simplicity of a shared log requires imposing a system-wide total order that is *expensive*, often *impossible*, and typically *unnecessary*. Previous work showed that a centralized, off-path sequencer can make such a total order feasible at intermediate scale (e.g., a small cluster of tens of machines) [7, 8]. However, at larger scale – in the dimensions of system size, throughput, and network bandwidth/latency – imposing a total order becomes expensive: ordering all updates via a sequencer limits throughput and slows down operations if machines are scattered across the network. In addition, for deployments that span geographical regions, a total order may be impossible: a network partition can cut off clients from the sequencer or a required quorum of the servers implementing the log. On the flip side, a total order is often unnecessary: updates to disjoint data (e.g., different keys in a map) do not need to be ordered, while updates that touch the same data may commute because the application requires weak consistency guarantees (e.g., causal consistency [5]). In this paper, we explore the following question: *can we provide the sim-*

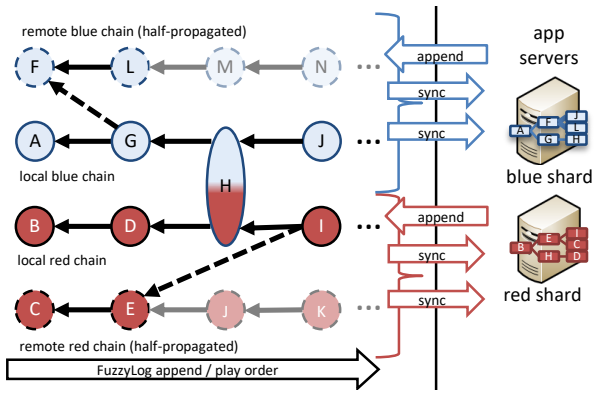


Figure 1: *In the FuzzyLog, each color contains updates to a data shard, while each chain contains updates from a geographical region.*

plicity of a shared log without imposing a total order?

We propose the FuzzyLog abstraction: a durable, iterable, and extendable order over updates in a distributed system. Crucially, a FuzzyLog provides a *partial order* as opposed to the total order of a conventional shared log. The FuzzyLog is a directed acyclic graph (DAG) of nodes representing updates to a sharded, geo-replicated system (see Figure 1). The FuzzyLog materializes a happens-after relation between updates: an edge from *A* to *B* means that *A* must execute after *B*.

The FuzzyLog captures two sources of partial ordering in distributed systems: data sharding and geo-replication. Internally, nodes in the FuzzyLog are organized into *colors*, where each color contains updates to a single application-level data shard. A color is a set of independent, totally ordered *chains*, where each chain contains updates originating in a single geographical region. Chains within a color are connected by cross-links that represent update causality. The entire DAG – consisting of multiple colors (one per shard) and chains within each color (one per region) – is fully replicated at every region and lazily synchronized, so that each region has the latest copy of its own chain, but some stale prefix of the chains of other regions. Figure 1 shows a FuzzyLog deployment with two data shards (i.e,. two colors) and two regions (i.e., two chains per color).

The FuzzyLog API is simple: a client can *append* a new node by providing a payload describing an update and the color of the shard it modifies. The new node is added to the tail of the local chain for that color, with outgoing cross-links to the last node seen by the client in each remote chain for the color. The client can *synchronize* with a single color, playing forward new nodes in the local region's copy of that color in a reverse topological sort order of the DAG. A node can be appended atomically to multiple colors, representing a transactional update across data shards.

Applications built over the FuzzyLog API are nearly as simple as conventional shared log systems. As shown in Figure 1, FuzzyLog clients are application servers that maintain in-memory copies or views of shared objects. To perform an operation on an object, the application appends an entry to the FuzzyLog describing the mutation; it then plays forward the FuzzyLog, retrieving new entries from other clients and applying them to its local view, until it encounters and executes the appended entry. The local views on the application servers constitute soft state that can be reconstructed by replaying the FuzzyLog. A FuzzyLog application that uses only a single color for its updates and runs within a single region is identical to its shared log counterpart; the FuzzyLog degenerates to a totally ordered shared log, and the simple protocol described above provides linearizability [23], durability, and failure atomicity for application state.

By simply marking each update with colors corresponding to data shards, FuzzyLog applications achieve scalability and availability. They can use a color per shard to scale linearly within a data center; transactionally update multiple shards via multi-color appends; obtain causal consistency [5] within a shard by using a color across regions; and toggle between strong and weak consistency when the network partitions and heals by switching between regions.

Implementing the FuzzyLog abstraction in a scalable and efficient manner requires a markedly different design from existing shared log systems. We describe Dapple, a system that realizes the FuzzyLog API over a collection of in-memory storage servers. Dapple scales throughput linearly by storing each color on a different replica set of servers, so that appends to a single color execute in a single phase, while appends that span colors execute in two phases (in the absence of failures) that only involve the respective replica sets. Dapple achieves this via a new fault-tolerant ordering algorithm that provides linear scaling for single-color appends, serializable isolation for multi-color appends, and failure atomicity. Across regions, a lazy synchronization protocol propagates each color's local chain to remote regions.

We implemented a number of applications over the FuzzyLog abstraction and evaluated them on Dapple. AtomicMap (201 lines of C++) is a linearizable, durable map that supports atomic cross-shard multi-puts, scaling to over 5.5M puts/sec and nearly 1M 2-key multi-puts/sec on a 16-server Dapple deployment. CRDTMap (284 LOC) provides causal+ consistency by layering a CRDT over the FuzzyLog. CAPMap (424 LOC) offers strong consistency in the absence of network partitions, but degenerates to causal+ consistency during partitions. We implemented a ZooKeeper clone over the FuzzyLog in 1881 LOC that supports linear scaling across shards and supports atomic cross-shard renames.

We also implemented a map that provides Red-Blue consistency [32], as well as a transactional CRDT [6].

Existing implementations of these applications are monolithic and complex; they often re-implement common mechanisms for storing, propagating, and ordering updates (such as protocols for atomic commit, consensus, and causality tracking). The FuzzyLog implements this common machinery efficiently under an explicit abstraction, hiding the details of protocol implementation while giving applications fine-grained control over sharding and geo-replication. As a result, applications can express different ordering requirements via simple invocations on the FuzzyLog API without implementing low-level distributed protocols.

Contributions: We propose the novel abstraction of a FuzzyLog (§3): a durable, iterable DAG of colored nodes representing the partial order of updates in a distributed system. We argue that this abstraction is *useful* (§4), describing and evaluating application designs that obtain the simplicity of the shared log approach while scaling linearly with atomicity, obtaining weaker consistency, and tolerating network partitions. We show that the abstraction is *practically feasible* (§5), describing and evaluating a scalable, fault-tolerant implementation called Dapple.

2 Motivation

The shared log approach makes distributed services simple to build by deriving properties such as durability, consistency, failure atomicity, and concurrency control via simple append/read operations on a shared log abstraction. We describe the pros and cons of this approach.

2.1 The simplicity of a shared log

In the shared log approach, application state resides in the form of in-memory objects backed by a durable, fault-tolerant shared log. In effect, an object exists in two forms: an ordered sequence of updates stored durably in the shared log; and any number of views, which are full or partial copies of the data structure in its conventional form – such as a tree or a map – stored in DRAM on clients (i.e., application servers). Importantly, views constitute soft state and are instantiated, reconstructed, and updated on clients as required by playing the shared log forward. A client modifies an object by appending a new update to the log; it accesses the object by first synchronizing its local view with the log.

As described in prior work [7, 8], this design simplifies the construction of distributed systems by extracting key properties via simple appends/reads on the shared log, obviating the need for complex distributed protocols. Specifically, the shared log is the source of consistency: clients implement state machine replication [46] by funneling writes through the shared log and synchronizing their views with it on reads. The shared log also provides durability: clients can recover views after crashes simply by replaying the shared log. It acts as a source of failure atomicity and isolation for transactions: the shared log is literally the serializable order of transactions.

2.2 The drawbacks of a total order

The shared log approach achieves a total order over all updates in a distributed system. We argue that such a total order can be *expensive* or *impossible* to achieve when services scale beyond the confines of a small cluster.

Total ordering is expensive. The traditional way to impose a total order is via a leader that receives updates from clients and sequences them; however, this limits the throughput of the system at the I/O bandwidth of a single machine [16]. CORFU [7] uses an off-path sequencer – instead of a leader – that issues tokens or contiguous positions in an address space to clients. To append data, a client first obtains a token from the sequencer – effectively reserving an address in the address space – and then writes the payload directly to a stripe of storage servers responsible for storing that address. This allows clients to totally order updates to a cluster of storage servers without pushing all I/O through a single machine; instead, the aggregate throughput of the system is limited by the speed at which the sequencer can update a counter and hand out tokens (roughly 600K ops/sec in CORFU [8]). To leverage the total order without requiring all clients to play back every entry, runtimes built over CORFU such as Tango [8] and vCorfu [57] support selective playback via streams and materialized streams, respectively. This requires sequencer state to be more complex than a single counter (e.g., per-stream backpointers [8] or additional stream-specific counters [57]).

While an off-path sequencer works well for small clusters (e.g., 20 servers in two adjacent racks [8]), it does not scale along a number of key dimensions. One such dimension is *network diameter*: since the sequencer lives in a fixed point in the network, far-away clients must incur expensive round-trips on each append. A second dimension is *network bandwidth*; sequencers are not I/O-bound or easily parallelizable, and cannot keep pace with recent order-of-magnitude increases in I/O bandwidth. On 1 Gbps networks, a sequencer that runs at 600K ops/s can support a 20-server CORFU deployment (1 Gbps per server or 30K 4KB appends/sec); however, on a 40 Gbps network, supporting 20 servers will require the sequencer to run at 24M ops/s. A third dimension is *payload granularity*: shared log applications do not store large payloads (in the limit, these could be 64-bit pointers to bigger items stored in some external blob store). With 100-byte payloads, the same sequencer will now have to run at nearly 1 billion ops/s. A final dimension is *system size*:

```
// constructs a new handle for playing a color
FL_ptr new_instance(colorID color, snapID snap=NULL);
// appends a node to a set of colors
int append(FL_ptr handle, char *buf, size_t bufsize,
      colorset *nodecolors);
// synchronizes with the log
snapId sync(FL_ptr handle, void (*callback)(char *buf,
      size_t bufsize));
// trims the color
int trim(FL_ptr handle, snapID snap);
```

Figure 2: *The FuzzyLog API.*

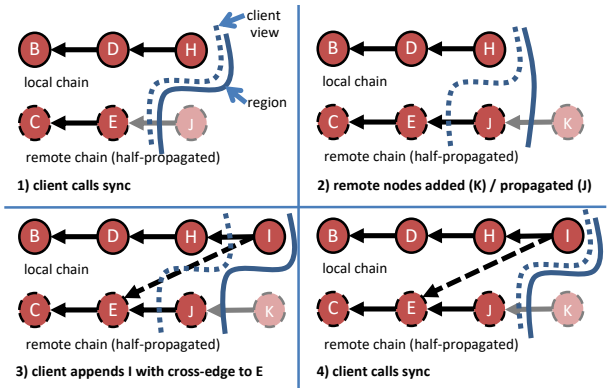


Figure 3: *The evolution of a single color.*

if we want to support 40 servers, we now need 2 billion ops/s from the sequencer.

Published numbers for sequencers in fully functional systems include: roughly 200K ops/sec (CORFU [7]), 250K ops/sec (NOPaxos [33]), and 600K ops/sec (Tango [8]). Stand-alone sequencers (i.e., simple counters without per-stream state) are faster; e.g., an RDMA-based counter runs at 122M ops/sec (80X faster than the next highest in the literature) [25]. Even at this speed, the largest cluster supported at 100 Gbps and a 512-byte payload would have just four servers.

Some approaches bypass the sequencer throughput cap at the cost of increasing append latency, either by aggressive batching [51] or writing out-of-order in the shared log address space and waiting for preceding holes to be filled [41]. The added append latency can be untenable for control plane services.

Total ordering is impossible. Regardless of how the total order is generated, it is fundamentally vulnerable to network partitions. Any protocol that provides a total order consistent with a linearizable order (i.e, if an update *B* starts in real time after another update *A* completes, then *B* occurs after *A* in the total order) is subject to unavailability during network partitions [14].

We find ourselves at a seeming impasse: a shared log enables simplicity and strong semantics for distributed systems, but imposes a total order that is expensive and sometimes impossible. We break this impasse with a partially ordered shared log abstraction.

3 The FuzzyLog Abstraction

The FuzzyLog addresses the ordering limitations in Section 2 via an expressive partial ordering API. The FuzzyLog's API captures two general patterns via which applications partially order operations. First, applications partition their state across logical data shards, such that updates against different shards are processed concurrently. Second, when deployed across geographical regions, applications weaken consistency to avoid synchronous cross-region coordination on the critical path of requests; as a result, updates across regions – even to the same logical data partition – can occur concurrently.

A FuzzyLog is a type of directed acyclic graph (DAG) that can be constructed and traversed concurrently by multiple clients. For clarity, we use the term 'node' exclusively to refer to nodes in the FuzzyLog DAG. Each node in the DAG is tagged with one or more colors. Colors divide an application's state into logical shards; nodes tagged with a particular color correspond to updates against the corresponding logical shard.

Each color is a set of totally ordered chains, one per region, with cross-edges between them that indicate causality. Every region has a full but potentially stale copy of each color; the region's copy has the latest updates of its own chain for the color, but stale prefixes of the other per-region chains for that color. Clients interact only with their own region's local copy of the DAG; they can modify this copy by appending to their own region's chain for a color.

Figure 2 shows the FuzzyLog API. A client creates an instance of the FuzzyLog with the `new_instance` call, supplying a single color to play forward. It can play nodes of this color with the `sync` call. It can `append` a node to a set of colors. We first describe the operation of these calls in a FuzzyLog deployment with a single color (i.e., an application with a single data shard).

The `sync` call is used by the client to synchronize its state with the FuzzyLog. A `sync` takes a snapshot of the set of nodes currently present at the local region's copy of a color, and plays all new nodes since the last `sync` invocation. Once all new nodes have been provided to the application via a passed-in callback, the `sync` returns with an opaque ID describing the snapshot. The nodes are seen in a reverse topological sort order of the DAG. Nodes in each chain are seen in the reverse order of edges in the chain. Nodes in different chains are seen in an order that respects cross-edges. Nodes in different chains that are not ordered by cross-edges can be seen in any order. Note that each node effectively describes a list of nodes – via its position in a totally ordered chain, and via

explicit pointers for cross-edges – that must be seen before it. Figure 3 shows the client synchronizing with the region in panel 1; trailing behind in panels 2 and 3; and synchronizing once again in panel 4. Snapshot IDs returned by `sync` calls at different clients can be compared to check if one subsumes the other.

When a client appends a node to a color with `append`, an entry is inserted into the local region's chain for that color. The entry becomes the new tail of the chain, and it has an edge in the DAG pointing to the previous tail; we define the tail as the only node in a non-empty chain with no incoming edge. The local region chain imposes a total order over all updates generated at that region. Further, outgoing cross-edges are added from the new node to the last node played by the client from every other per-region chain for the color. In effect, the newly appended node is ordered after every node of that color seen by the client. For example, in Figure 3 (panel 3), a client appends a new node I to the region's local chain (after node H), with a cross-edge to E, which is the latest node in the remote chain seen by the client.

To garbage collect the FuzzyLog, clients can call `trim` on a snapshot ID to indicate that the nodes in it are no longer required (e.g., because the client stored the corresponding materialized view in some durable external store). A snapshot ID can also be provided to the `new_instance` call, in which case playback skips nodes within the snapshot; this allows a new client to join the system without playing the FuzzyLog from the beginning.

While the `sync` and `trim` calls operate over a single color, the FuzzyLog supports appending to multiple colors. An `append` to a set of colors atomically appends the entry to the local chains for each color. The new node is reflected by `sync` calls on any one of the colors involved. If a node is in multiple colors, trimming it in one color does not remove it from the other colors it belongs to.

Semantics: Operations to a single color across regions are causally consistent. In other words, two `append` operations to the same color issued by clients in different regions are only ordered if the node introduced by one of them has already been seen by the client issuing the other one. In this case, an edge exists in the DAG from the second node to the first one. The internal structure of the DAG ensures that the copies at each region converge even though concurrent updates can be applied in different orders to them: since the clients at each region modify a disjoint part of the DAG (i.e., they append to their own per-region chain), there are never any conflicts when the copies are synchronized.

Operations within a single region are serializable. All `append` and `sync` operations issued by clients within a region execute in a manner consistent with some serial execution. This serialization order is linearizable if the operations are to a single color within the region (i.e., on a single chain); it does not necessarily respect real-time ordering when `append` operations span multiple colors.

Discussion: Designing the FuzzyLog API required balancing the power of the API against its simplicity and the feasibility of implementing it. In earlier candidates for the API, we directly exposed chains to programmers and allowed `append`/`sync` on any subset of them with a choice of consistency guarantees. This API rendered a scalable implementation much more difficult; for example, guaranteeing a topological sort order for nodes in a subset of chains required us to potentially traverse every chain in the system. In addition, the consistency choices required programmers to reason about the performance and availability of different combinations (e.g., strongly consistent multi-appends on chains in different regions can block due to network partitions). We were able to drastically simplify the API once we realized the equivalence between colors and shards: for example, it makes sense for clients to play a single color since doing otherwise negates the scaling benefit of sharding; and to obtain causal consistency within a color since it is geo-replicated across regions that can partition.

4 FuzzyLog Applications

This section describes how applications can use the FuzzyLog API with a case study of an in-memory key-value storage service. In this section, the term 'server' refers exclusively to application servers storing in-memory copies of the key-value map, which in turn are FuzzyLog clients. We start with a simple design called LogMap that runs over a single color within a single region (i.e., it effectively runs over a single totally ordered shared log). Each LogMap server has a local in-memory copy of the map and supports `put`/`get`/`delete` operations on keys. The server continuously executes a `sync` on the log in the background and applies updates to keep its local view up-to-date. A `get` operation at the server simply waits for a `sync` to complete that started after it was issued, before accessing the local view and returning; this ensures that any updates that were appended to the FuzzyLog before the `get` was issued are reflected in the local view, providing linearizability. A `put`/`delete` operation appends a node to the FuzzyLog describing the update; it then waits for a `sync` to apply the update to the local view, at which point it returns.

This basic LogMap design – implemented in just 193 lines of code – enables durability, high availability, strong consistency, concurrency control and failure atomicity. It is identical to previously described designs [7] over a conventional shared log. However, its reliance on a single total order comes at the cost of scalability, performance, and availability. The remainder of this section describes how LogMap can be modified to

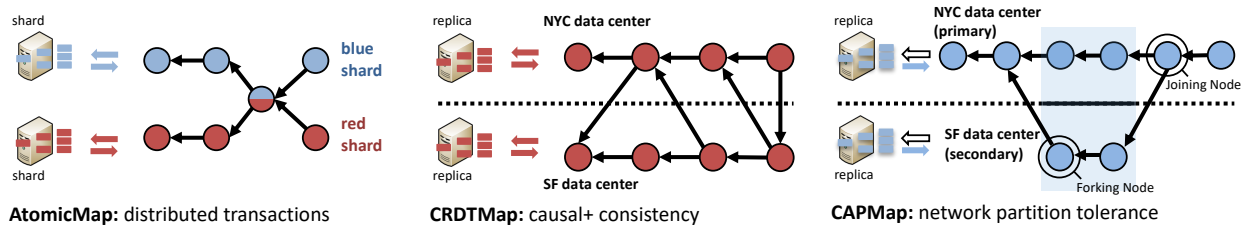


Figure 4: *FuzzyLog capabilities: AtomicMap, CRDTMap, and CAPMap.*

use the FuzzyLog to circumvent each of these limitations.

4.1 Scaling with atomicity within a region

We first describe applications that run within a single region and need to scale linearly. In ShardedMap (193 LOC), each server stores a shard of the map; each shard corresponds to a FuzzyLog color. Updates to a particular shard are appended as nodes of the corresponding color to the FuzzyLog; each server syncs its local state with the color of its shard. This simple change to LogMap – requiring just the color parameter to be set appropriately on calls to the FuzzyLog – provides linear scalability for linearizable `put`/`get` operations.

The FuzzyLog supports atomicity across shards. If the atomic operation required is a simple blind `multi-put` that doesn't return a value, all we require is a simple change to append an update to a set of colors instead of a single one, corresponding to the shards it modifies. AtomicMap (201 LOC, Figure 4 (Left)) realizes this design. One subtle point is that since FuzzyLog multi-color appends are serializable, AtomicMap is also serializable, not linearizable or strictly serializable.

To implement read/write transactions with stronger isolation levels, we use a protocol identical to the one used by Tango [8]. In TXMap (417 LOC), each server executes read-write transactions speculatively [8, 10], tracking read-sets and buffering write-sets. To commit, the server appends a speculative intention node into the FuzzyLog to the set of colors corresponding to the shards being read and written. When a server encounters the intention node in the color it is playing, it appends a second node with a yes/no decision to the set of colors. To generate this decision, the server examines the sub-part of the transaction touching its own shard and independently (but deterministically) validates it (e.g., checking for read-write conflicts when providing strict serializability). A server only applies the transaction to its local state if it encounters both the original intention and a decision marked yes for each color involved.

Interestingly, this protocol provides strict serializability even though the FuzzyLog itself is only serializable. Intuitively, within a single color, if a client waits after appending an intention for a transaction T until it plays the node, it is guaranteed to have seen all transactions that could appear before T in the serial order. As a result, future transactions must appear later in the serial order, ensuring strict serializability. In a multi-color transaction, we need to ensure that all transactions in all the colors involved that could appear before T have been seen. A decision node conveys two things: that all such transactions in a color have been seen; and whether they conflict with the transaction. As in Tango [8], our protocol requires at least one application server to be available for each shard in order to generate decision records.

4.2 Weaker consistency across regions

Applications can often tolerate weaker consistency guarantees. One example is causal consistency [5], which roughly requires the following: if a server performs an update U_1 after having seen an update U_0, then any other server in the system must see U_0 before U_1. If U_1 and U_2 were performed independently by servers that did not see each other's update, they can be seen in any order.

CRDTMap implements a causally consistent map. In Figure 4 (Middle), the map is replicated across two regions, one in NYC and another in SF. CRDTMap simply uses a single color for all updates to a map; in each region, put operations are appended to the local chain for the color and propagated asynchronously to the other region. Since the partial order within a color is exactly the causal order of updates, each server playing the color observes updates in a causally consistent order.

To achieve convergence when servers see causally independent updates in different orders, we employ a design for CRDTMap based on the Observed-Remove set CRDT [47], which exploits commutativity to execute concurrent updates in conflicting orders without requiring rollback logic. The CRDT design achieves this by predicating deletions performed by a server on put operations that the server has already seen; accordingly, each delete node in the DAG lists the put operations that it subsumes.

4.3 Tolerating network partitions

While CRDTMap can provide availability during network partitions, it does so by sacrificing consistency even when there is no partition in the system. CAPMap (named after the CAP conjecture [14]) provides strong consistency in the absence of network partitions and causal consistency during them (see Figure 4 (Right)).

As with our other map designs, CAPMap appends entries on `put` operations and then syncs until it sees the appended node. Unlike them, CAPMap requires servers to communicate with each other, albeit in a simple way: servers route FuzzyLog appends through proxies in other regions. To perform a `put` in the absence of network partitions, the server routes its append through a proxy in a primary region; it then syncs with its own region's copy of the FuzzyLog until it sees the new node, before completing the `put`. As a result, a total order is imposed on all updates (via the primary region's chain for the color), and the map is linearizable.

When a secondary region is partitioned away from the primary region, servers switch over to appending to the FuzzyLog in the local region, effectively 'forking' the total order. CAPMap sets a flag on these updates to mark them as secondary nodes (i.e., appends occurring at the secondary). Updates that were in-flight during the network partition event may be re-appended to the local region, appearing in the DAG as identical nodes in the primary and secondary forks. When the network partition heals, servers at the secondary stop appending locally and resume routing appends through the proxy at the primary. Every routed append includes the snapshot ID of the last `sync` call at the secondary client; the proxy blocks the append until it sees a subsuming snapshot ID on a `sync`, ensuring that all the nodes seen by the secondary client have also been seen by the proxy and are available at the primary region.

The FuzzyLog explicitly captures the effects of a network partition, including concurrent activity in the regions and duplicate updates. As a result, CAPMap can relax and reimpose strong consistency via a simple playback policy over the FuzzyLog. Any server playing the DAG after the partition heals enforces a deterministic total order over nodes in the forked section: when it encounters any secondary nodes, it buffers them until the next primary node (i.e., the joining node). All buffered nodes are then applied immediately before the joining node (ignoring duplicate updates), ensuring that all servers observe the same total order and converge to the same state.

Secondary servers that experience a network partition continue operating over the local fork, applying changes to a speculative copy of state. When the partition heals, each secondary server throws away its speculative changes after the forking node and replays the nodes in the forked region of the DAG, applying updates in the primary fork before re-applying the secondary fork. Our CAPMap implementation realizes this speculative copy by cloning state on a fork, and throwing away the clone when the partition heals; but more efficient copy-on-write mechanisms could be used as well.

As a result, we obtain causal+ consistency [35] during network partitions and linearizability otherwise. Importantly, CAPMap achieves these properties via simple append and playback policies over the structure and contents of the FuzzyLog.

4.4 Other designs

TXCRDTMap: Two properties discussed so far – transactions within a single region and weak consistency across regions – can be combined to provide geo-distributed transactions. By changing 80 LOC in CRDTMap, we can obtain a transactional CRDT that provides cross-shard failure atomicity [6] (or equivalently, an isolation guarantee similar to Parallel Snapshot Isolation [48]).

RedBlueMap: The FuzzyLog can support RedBlue consistency [32], in which blue operations commute with each other and with all red operations, while red operations have to be totally ordered with respect to each other, but not blue operations. RedBlue consistency can be implemented with a single color. One of the regions is designated a primary, and 'Red' operations are routed to the primary via a proxy (and thus totally ordered, similar to CAPMap). 'Blue' operations are performed at the local region. We implemented RedBlueMap in 330 LOC.

COPSMap: While CRDTMap can be scaled by sharding system state across different per-color instances, an end-client interacting with such a store will not get causal consistency across shards [35, 36]. Concretely, in a system with two regions and two colors, an end-client in one region may issue a put on a red server, and subsequently issue a put on a blue server. Once the blue put propagates to the remote region, a different end-client may issue a get on a blue server, and subsequently a get on a red server. If the end-client sees the blue put, it must also see the red put, since they are causally related. To provide such a guarantee, the map server can return a snapshot ID with each operation; the end-client can maintain a set of the latest returned snapshot IDs for each color and provide it to the map server on each operation, which in turn can include it in the appended node. In such a scheme, when the blue server in the remote region sees the blue put, it contacts a red server to make sure the causally preceding red node has been seen by it and exists in the region. Such a design requires servers playing different colors to gossip the last snapshot IDs they have seen for their respective colors. We leave the

COPSMap implementation for future work.

4.5 Garbage collection

As with shared log systems, GC is enormously simplified by the nature of the workload: the log is used to store a history of commands rather than first-class data, and can be trimmed in increasing prefixes. At any time, the application can store its current in-memory state (and the associated snapshot ID) durably on some external storage system, or alternatively ensure that enough application servers have a copy of it. Once it does so, it can issue the `trim` command on the snapshot ID. Clients that are lagging behind may encounter an `already_trimmed` error, in which case they must retrieve the latest durable state from the external store, and then continue playing the log from that point.

5 Dapple Design / Implementation

Dapple is a distributed implementation of the FuzzyLog abstraction, designed with a particular set of requirements in mind. The first is *scalability*: reads and appends must scale linearly with the number of colors used by the application and the number of servers deployed by Dapple, assuming that load is balanced evenly across colors. The second requirement is *space efficiency*: the FuzzyLog partial order has to be stored compactly, with edges represented with low overhead. A third requirement is *performance*: the `append` and `sync` operations must incur low latency and I/O overhead.

Dapple implements the FuzzyLog abstraction over a collection of storage servers called chainservers, each of which stores multiple in-memory log-structured address spaces. Dapple partitions the state of the FuzzyLog across these chainservers: each color is stored on a single partition. Each partition is replicated via chain replication [54]. Our current implementation assumes for durability that storage servers are outfitted with battery-backed DRAM [17, 24]. We first describe operations against a single color on an unreplicated chainserver.

5.1 Single-color operation

Recall that each FuzzyLog color consists of a set of totally ordered chains, one per region; each region has the latest copy of its own local chain, but a potentially stale copy of the other regions' chains. Dapple stores each chain on a single log, such that the order of the entries in the log matches the chain order (i.e., if a chain contains an edge from B to A, B appears immediately after A in the corresponding log). In a deployment with R regions, each region stores R logs, one per chain. Clients in the region actively write to one of these (the local log), while the remaining are asynchronously replicated from other regions (we call these shadow logs). Each server exposes a low-level API consisting of three primitives: `log-append`, which appends an entry to a log; `log-snapshot`, which accepts a set of logs and returns their current tail positions; and `log-read`, which returns the log entry at a given position.

Clients implement the `sync` on a color via a `log-snapshot` on the logs for that color, followed by a sequence of `log-reads`. The return value of `log-snapshot` acts as a vector timestamp for the color, summarizing the set of nodes present for that color in the local region; this is exactly the snapshot ID returned by the `sync` call. The client library fetches new nodes that have appeared since its last `sync` via `log-read` calls. When the application calls `append` on a color, the client library calls `log-append` on the local log for that color. It includes the vector timestamp of nodes seen thus far in the new entry; as a result, each appended entry includes pointers to the set of nodes it causally depends on (these are the cross-edges in the FuzzyLog DAG). On a `sync`, the client library checks each entry it reads for dependencies and recursively fetches them before delivering them to the application. In this manner, the client ensures that playback of a single color happens in DAG order.

Each chainserver periodically synchronizes with its counterparts in remote regions, updating the shadow logs with new entries that originated in those regions. To fetch updates, the chainserver itself acts as a client to the remote chainserver and uses a `sync` call; this ensures that cross-chain dependencies are respected when it receives remote nodes. Copied-over entries are reflected in subsequent `sync` calls by clients and played; new entries appended by the clients then have cross-edges to them.

Dapple replicates each partition via chain replication. Each `log-append` operation is passed down the chain and acknowledged by the tail replica, while `log-snapshot` is sent directly to the tail. Once the client obtains a snapshot, subsequent `log-read` operations can be satisfied by any replica in the chain. The choice of replication protocol is orthogonal to the system design: we could equally use Multi-Paxos.

5.2 Multi-color operation

The FuzzyLog API supports appending a node to multiple colors. In Dapple, this requires atomically appending a node to multiple logs: one log per color corresponding to its local region chain. To do so, Dapple uses a classical total ordering protocol called Skeen's algorithm (which is unpublished but described verbatim in other papers, e.g., Section 4 in Guerraoui et al. [22]) to consistently order appends.

Skeen's original algorithm produces a serializable order for operations by multiple clients across different subsets of servers. Unfortunately, it is not tolerant to the failure of its participants. In our setting, each 'server' is a replicated partition of chainservers and can be as-

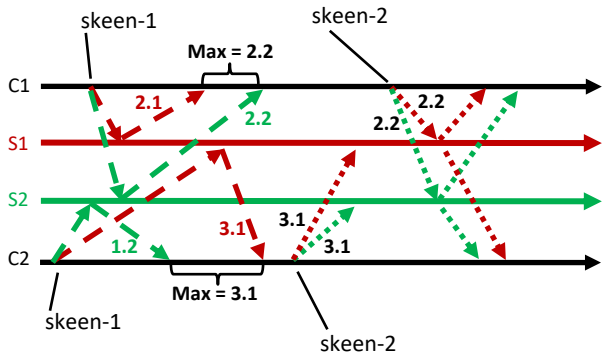


Figure 5: *Distributed ordering for multi-appends: servers return timestamps X.Y in phase 1 where X is a local logical clock and Y is a server-specific nonce.*

sumed to not fail. However, the clients in our system are unreplicated application servers that can crash. We assume that such client failures are infrequent; this pushes us towards a protocol that is fast in the absence of client failures and slower but safe when such failures do occur. Accordingly, we add three fault-tolerance mechanisms – leases, fencing, and write-ahead logging – to produce a variant of Skeen's that completes in two phases in a failure-free 'fast' path, but can safely recover if the origin client crashes.

Each chainserver maintains a local logical Lamport clock [28]. All client operations are predicated on relatively coarse-grain leases [20] (e.g., 100 ms), which they obtain from each server (or the head of the replica chain for each partition); if the lease expires, or the head of the replica chain changes, the operation is rejected.

We now describe failure-free operation. The fast path consists of two phases, and has to execute from start to completion within the context of a single set of leases, one per involved partition. For ease of exposition, we assume each partition has one chainserver replica.

In the first phase, an origin client (i.e., a client originating a multi-append) contacts the involved chainservers, each of which responds with a timestamp consisting of the value of its clock augmented with a server-specific unique nonce to break ties. Each chainserver inserts the multi-append operation into a pending queue along with the returned timestamp. For example, in Figure 5, origin client C1 contacts S1, which responds with 2.1, where the local clock value is 2 and the unique nonce is 1. In addition, the origin client provides a WAL (write-ahead log) entry that each chainserver stores; this includes the payload, the colors involved, and the set of leases used by the multi-append.

Once the client hears back from all the involved chainservers, it computes the max across all received timestamps, and transmits that back to the chainservers in a second phase: this max is the timestamp assigned to the multi-append and is sufficient to serialize the multi-appends in a region. For example, in Figure 5, client C1 sends back a max timestamp of 2.2 to servers S1 and S2. When a chainserver receives this message, it moves the multi-append from the pending queue to a delivery queue; it then waits until there is no other multi-append in the pending queue with a lower returned timestamp, or in the delivery queue with a lower max timestamp (i.e., no other multi-append that could conceivably be assigned a lower max timestamp). Once this condition is true, the multi-append is removed from the delivery queue and processed. In Figure 5, server S1 receives a phase 2 message with a max timestamp of 3.1 from client C2, but does not respond immediately since it previously responded to a phase 1 message from client C1 with a timestamp of 2.1. Once C1 sends a phase 2 message with a max timestamp of 2.2, S1 knows the ordering for both outstanding multi-appends and can respond to both C1 and C2.

The protocol described above completes in two phases. A third step off the critical path involves the client sending a clean-up message to delete the per-append state (the WAL, plus a status bit indicating the last executed phase) at the chainservers; this is lazily executed after a multiple of the lease time-out, and can be piggybacked on other messages. If a lease expires before the two phases are executed at the corresponding server, or the origin client crashes, it leaves one or more servers in a wedged state, with the multi-append stuck in the pending queue and blocking new appends to the colors involved. After a time-out, the chainserver begins responding to new append requests with a *stuck-err* error message, along with the WAL entry of the stuck multi-append. A client that receives such an error message can initiate the recovery protocol for the multi-append.

A client recovering a stuck multi-append (i.e., a recovery client) proceeds in three phases: it fences activity by the origin client or other recovery clients; determines the wedged state of the system; and completes the multi-append. The fencing phase involves accessing the lease set of the original client (which is stored in the WAL), invalidating it at the servers, and writing a new recovery lease set at a designated test-and-set location on one of the chainservers. If some other recovery client already stored a lease set at this location, we wait for that client to recover the append, fencing it after a time-out. Fencing ensures that at any given point, only one client is active; the WAL allows clients to deterministically roll forward the multi-append.

Correctness: Skeen's protocol has been proven to generate a total order by others [22, 45]. To prove our recovery protocol correct, we wrote conventional proofs as well as a machine-checked proof in Coq. We omit the

full proof for lack of space. Informally, we prove that the test-and-set mechanism ensures that only one client is actively mutating the state of the system at any given point in time. Further, we show that each append can be modeled as a four-stage state machine (some servers in phase 1, some uninitiated; some in phase 1, some in phase 2; some in phase 2, some completed; all completed). Any recovery client finds the system in a particular state and advances it in a manner identical to the non-failing case.

Performance and availability: The append protocol takes two phases in the fast path and three in the recovery path. The protocol can block if the logs being appended to reside on different sides of a network partition; however, the semantics of colors in FuzzyLog ensure that we only append to logs within a single region. Single-color appends follow the same protocol as multi-appends, but complete in a single phase that compresses the two phases of the fast path.

A subtle point is that a missed fast path deadline will block other multi-appends from completing, but will not cause them to miss their own deadlines; they are free to complete the fast path and receive a timestamp, and only block in the delivery queue. As a result, a crashed client will cause a latency spike but not a cascading series of recoveries. In addition, this protocol is subject to FLP [18] and susceptible to livelock, since recovery clients can fence each other perpetually. Our implementation mitigates this by having clients back-off for a small, randomized time-out if they encounter an ongoing recovery, before fencing it and taking over recovery.

6 Evaluation

We run all our experiments on Amazon EC2 using *c4.2xlarge* instances (8 virtual cores, 15 GiB RAM, Intel Xeon E5-2666 v3 processors). Most of the experiments run within a single EC2 region; for geo-distributed experiments, we ran across the us-east-2 (Ohio) and the ap-northeast-1 (Tokyo) regions, which are separated by an average ping latency of 168ms. In all experiments, we run Dapple with two replicas per partition unless otherwise specified. All throughput numbers are without any application-level batching.

We first report latency micro-benchmarks for Dapple on a lightly loaded deployment. Figure 6 shows the distribution of latencies for 16-byte appends involving one color (top) and two colors on different chainservers (middle), as well as the latency to recover stuck multi-appends due to crashed clients (bottom). In all cases, latency increases with increasing replication factor due to chain replication. At every replication factor, single-color appends are executed with lower latency than two-color appends, which in turn require lower latency than two-color recovery. This difference in latency arises because single-color appends execute in a single phase,

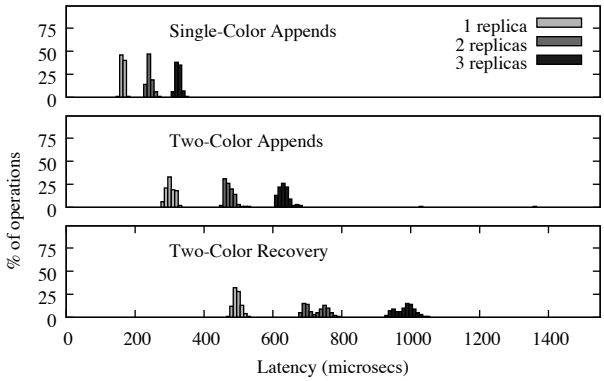


Figure 6: *Dapple executes single-color appends in one phase; multi-color appends in two phases; and recovers from crashed clients in three phases.*

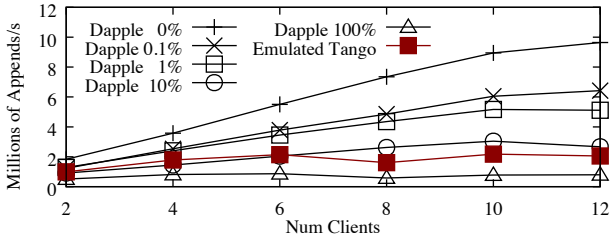


Figure 7: *Dapple scales with workload parallelism, but a centralized sequencer bottlenecks emulated Tango.*

while two-color appends execute in two phases and two-color recoveries execute in three phases.

The remainder of our evaluation is structured as follows: First, we evaluate the differences between Dapple and prior shared log designs (§6.1). Second, we use the Map variants from §4 to show that Dapple provides linear scaling with atomicity (§6.2), weaker consistency guarantees (§6.3), and network partition tolerance (§6.4). Finally, we describe a ZooKeeper clone over Dapple (§6.5).

6.1 Comparison with shared log systems

In this experiment, we show that centralized sequencers in existing shared log systems fundamentally limit scalability. Shared log systems such as Tango [8] and vCorfu [57] use a centralized sequencer to determine a unique monotonic sequence number for each append. Based on its sequence number, each append is deterministically replicated on a different set of servers. The sequencer therefore becomes a centralized point of coordination, even when requests execute against different application-level data-structures or shards. In contrast, Dapple allows applications to naturally express their sharding requirements via colors, and can execute appends to disjoint sets of colors independently.

We emulate Tango's append protocol in Dapple by us-

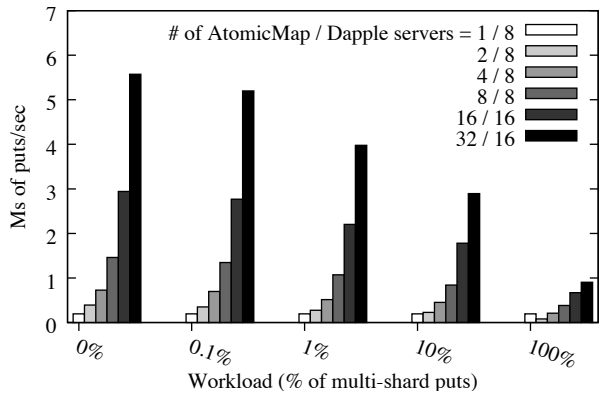


Figure 8: *AtomicMap scales throughput while supporting multi-shard transactions. Each bar labelled N / K shows throughput with N AtomicMap servers running against a K-server Dapple deployment.*

ing five chainserver partitions to store data, and a single unreplicated server to disperse sequence numbers; given a sequence number, appends are deterministically written (via a Dapple-`append`) to one of the five chainserver partitions in a round-robin fashion. We compare this to a FuzzyLog deployment that uses five chainserver partitions. The number of partitions and replication factor in emulated Tango and Dapple are identical, while emulated Tango uses an extra server for sequencing. We run a workload where each client appends to a particular color, mixing single-color appends with a fixed percentage of appends that include a second, randomly picked color. Figure 7 shows average throughput over a 10-second run for workloads with different percentages of two-color appends. Emulated Tango cannot scale beyond four clients due to its use of a centralized sequencer. Dapple scales near-linearly when the workload is fully partitionable (0% multi-color appends), is 2X faster at 1% multi-color appends, and matches Tango at 10% multi-color appends. At 100% multi-color appends, Dapple performs worse because the required partial order is nearly a total order, which Tango provides more efficiently.

6.2 Scalable multi-shard atomicity

The FuzzyLog allows applications to scale within a region by sharding across colors, and supports multi-shard transactions via multi-color appends. We now demonstrate the scalability of an AtomicMap (Section 4.1), which partitions its state across multiple colors. Each AtomicMap server is a Dapple client, and is affinitized with a unique color (corresponding to a logical partition of the AtomicMap's state). Each client performs a combination of single puts against its local partition and multi-puts against its partition and a randomly selected remote partition.

Figure 8 shows the results of the AtomicMap experiment. For different percentages of multi-puts in the workload (on the x-axis), we vary system size and plot throughput on the y-axis. We use between 8 and 16 chainservers in Dapple (deployed without replication since we ran into EC2 instance limits). We use 8-byte keys and 8-byte values to emulate a workload where the AtomicMap acts as an index storing pointers to an external blob store. Keys for put operations are selected uniformly at random from a key space of 1M keys.

Figure 8 shows that under 0% multi-shard puts, throughput scales linearly from 1 to 16 AtomicMap servers. The throughput jump from 16 to 32 servers is slightly less than 2x because we pack two Dapple clients per AtomicMap server at the 32 client data point (due to the EC2 instance limit). As the percentage of multi-shard puts increases from 0.1% to 100%, scalability and absolute throughput degrade gracefully. This is expected due to the extra cost of executing multi-shard puts (each requires a two-phase multi-color append).

6.3 Weaker consistency guarantees

Dapple allows geo-distributed applications to perform updates to the same color with low latency. By composing a single color out of multiple totally ordered chains, one per geographical region, a client in a particular region can append updates to a color without performing any coordination across regions in the critical path. This section demonstrates this capability via a CRDTMap.

In Figure 9, we host a single, unpartitioned CRDTMap on five application servers (i.e., Dapple clients); we locate each in a virtual region with its own Dapple copy, all running in the same EC2 region. Four of these servers are writers issuing put operations at a controlled aggregate rate (left y-axis), while the fifth is a reader issuing get operations on the CRDTMap. Each writing server uses four writer processes. The gets observe some frontier of the underlying DAG, and can therefore lag behind by a certain number of puts (right y-axis), but are fast, local operations. Midway through the experiment, we spike the put load on the system; this does not slow down get operations at the reader (not shown in the graph), but instead manifests as staleness.

6.4 Network partition tolerance

Dapple allows applications to provide strong consistency during normal operation and weak consistency under network partitions. In this experiment, we demonstrate this capability by running CAPMap across a primary and a secondary region (us-east-2 and ap-northeast-1, respectively). The experiment lasts for 14 seconds. From 0-6 seconds, the primary and secondary regions are connected. Between 6-8 seconds, we simulate a network

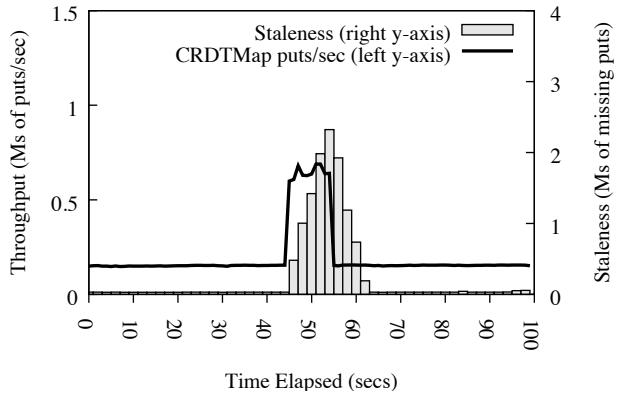


Figure 9: *CRDTMap provides a trade-off between throughput and staleness.*

partition between the primary and secondary. Finally, from 8-14 seconds, connectivity between the primary and secondary is restored. Each region runs two servers, one issuing puts and one issuing gets. We measure the latency of gets and puts (y-axis), against the wall-clock time they are issued at (x-axis).

Figure 10 shows the results of the experiment. In normal operation (0 to 6 seconds), all updates are stored in a single primary chain, and both regions get strong consistency; the secondary has high latencies for puts and gets due to the 168 ms inter-region roundtrip it incurs to access the primary chain. At 6 seconds, the network between the regions partitions; the primary continues to obtain strong consistency and low latency, but the secondary switches to weaker consistency, storing its updates on a local secondary chain (and obtaining much lower latency for puts/gets in exchange for the weaker consistency). At 8 seconds, the network heals; the secondary appends a joining node to the primary chain via a proxy in the primary region. As part of this joining request, the secondary provides a snapshot ID reflecting the last node it appended to its local chain. The proxy at the primary waits until the nodes in the snapshot are replicated to the primary region and seen by it before completing the joining append. The joining append causes a high latency put by the secondary just after the partition heals, and a spike in get latency on the primary as it plays nodes appended to the secondary chain during the partition.

6.5 End-to-end applications

We implemented a ZooKeeper clone, DappleZK in 1881 LOC of Rust. DappleZK partitions a namespace across a set of servers, each of which acts as a Dapple client, storing a partition of the namespace in in-memory data-structures backed by a FuzzyLog color.

This section compares DappleZK's performance with

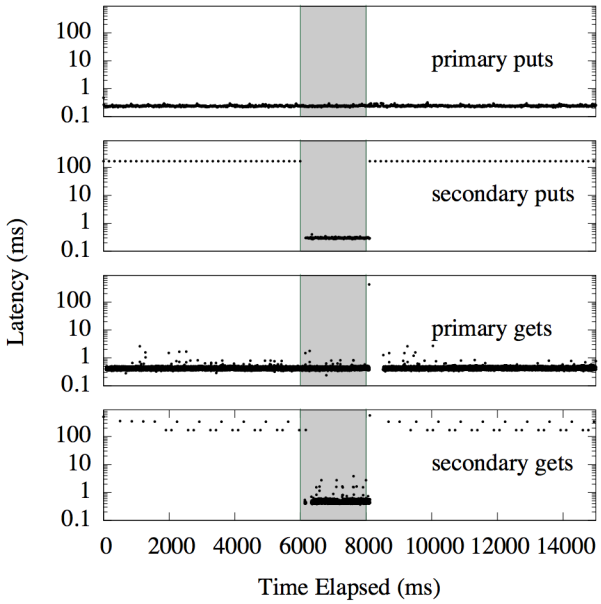


Figure 10: *CAPMap switches between linearizability and causal+ consistency during network partitions.*

ZooKeeper. Each DappleZK server is responsible for an independent shard of the ZooKeeper namespace, and atomically creates and renames files. Create operations are restricted to a single DappleZK shard. Each rename atomically moves a file from one DappleZK shard to another via the distributed transaction protocol described in Section 4.1.

We partition the ZooKeeper namespace across 12 DappleZK shards, and run one DappleZK server per shard. We deploy Dapple with either one or two partitions. Each partition is configured with three replicas. DappleZK uses two coloring schemes; a color per parti-

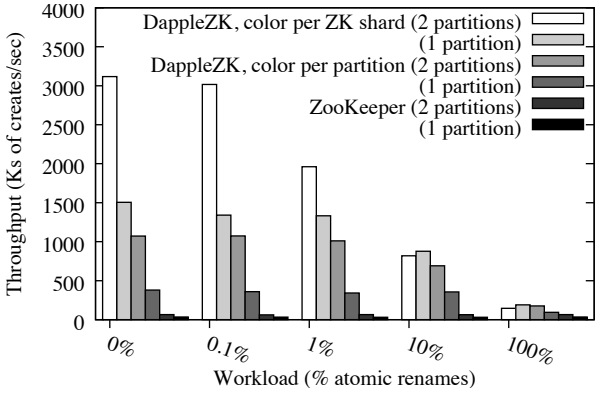


Figure 11: *DappleZK exploits Dapple's partial ordering to implement a scalable version of the ZooKeeper API.*

tion and a color per DappleZK shard. In the color per partition deployment, each color holds updates corresponding to multiple DappleZK server shards.

We run conventional ZooKeeper with three replicas, and also include a partitioned ZooKeeper deployment with two partitions. Our ZooKeeper deployments keep their state in DRAM to enable a fair comparison. Note that ZooKeeper does not support atomic renames; we emulated renames on it by executing a delete and create operation in succession. We include the ZooKeeper comparison for completeness; we expect the FuzzyLog single-partition case to outperform ZooKeeper largely due to the different languages used (Rust vs. Java) and the difference between prototype and production-quality code.

Figure 11 shows the results of the experiment. We vary the percentage of renames in the workload on the x-axis, and plot throughput on the y-axis. Each x-axis point shows a cluster of bars corresponding to the four DappleZK configurations and two ZooKeeper configurations. With a single color and a single partition, every DappleZK server stores its state on the same color. DappleZK servers perform their appends and reads against the same color, which limits their throughput. With two partitions, the number of DappleZK servers per color is halved, which increases throughput. When we switch to a color per DappleZK server, throughput increases dramatically because requests from different DappleZK servers do not need to be serialized against the same color. The addition of another partition further increases throughput because the colors can be spread across two partitions. When deployed with a single partition, Dapple servers were overloaded, which led to extra scheduling overhead and caused the two partition case to outperform a single partition by over 2X (in both color per ZK shard and color per partition cases). With an increasing fraction of atomic renames, throughput decreases because DappleZK must perform a distributed transaction across the involved DappleZK servers. In comparison to DappleZK, ZooKeeper provided 36K and 66K ops/s with one and two partitions respectively.

7 Related Work

Abstractions for ordering updates in a distributed system have a long history. Examples include Virtual Synchrony [13, 53], State Machine Replication [46], Viewstamp Replication [42], Multi-Paxos [52], and newer approaches such as Raft [43]. Most of these impose a total order on updates; the exceptions track particular partial orders imposed by operation commutativity (pessimistically [30, 37] and optimistically [26]), causal consistency (as in Virtual Synchrony and Lazy Replication [27]), or network partitions (as in Extended Virtual Synchrony [38]). In contrast, the FuzzyLog expresses the partial orders relating to both causality and data sharding within a single ordering abstraction.

FuzzyLog designs for providing weaker consistency are informed by a number of systems: COPS [35] and Eiger [36] provide causal consistency in a partitioned store, while Bayou allows for disconnected updates and eventual reconciliation [44, 49]. TARDiS [15] exposes *branch-on-conflict* as an abstraction in a fully replicated, multi-master store. In contrast to the TARDiS DAG, the FuzzyLog allows applications to construct a wider range of partial orders (e.g., CAPMap branches on network partitions rather than conflicts), and enables distributed transactions via color-based partitioning.

A number of systems provide distributed transactions over addresses or objects [4, 34]. Recent systems leverage modern networks such as RDMA and Infiniband to enable high-speed transactions [17, 31]. FuzzyLog provides a lower layer of abstraction, which in turn supports general-purpose transactions using shared log techniques [8, 10]. There has also been recent interest in improving distributed transaction throughput and latency via techniques such as transaction chopping [39, 58, 59, 61]. These mechanisms could be employed by transactional FuzzyLog applications.

Finally, the FuzzyLog is heavily inspired by shared log designs from research [7, 8, 10] and industry [1, 2, 55].

8 Conclusion

The shared log approach simplifies the construction of control plane services, but tightly bounds the scalability and consistency of the resulting systems. The FuzzyLog abstraction – and its implementation in Dapple – extends the shared log approach to partial orders, allowing applications to scale linearly without sacrificing transactional guarantees, obtain a range of consistency guarantees, and switch seamlessly between these guarantees when the network partitions and heals. Crucially, applications can achieve these capabilities in hundreds of lines of code via simple, data-centric operations on the FuzzyLog, retaining the core simplicity of the shared log approach.

Acknowledgments

This work was funded primarily by an NSF AitF grant (CCF-1637385), and partly by NSF grants CCF-1650596 and IIS-1718581. We thank Zhong Shao for his significant input from the beginning of the project. We also thank Luis Rodrigues and Yair Amir for feedback on the ideas behind the FuzzyLog. Vijayan Prabhakaran and Hakim Weatherspoon provided valuable comments on early drafts of this paper. Finally, we would like to thank Kang Chen for shepherding the paper, as well as the anonymous reviewers for their insightful reviews.

References

[1] Facebook logdevice. https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/.

[2] VMware CorfuDB. https://github.com/CorfuDB/CorfuDB.

[3] Zlog transactional key-value store. http://noahdesu.github.io/2016/08/02/zlog-kvstore-intro.html.

[4] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SOSP 2007*.

[5] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995), 37–49.

[6] AKKOORATH, D. D., TOMSIC, A. Z., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIÇA, N., AND SHAPIRO, M. Cure: Strong semantics meets high availability and low latency. In *IEEE ICDCS 2016*.

[7] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. Corfu: A shared log design for flash clusters. In *USENIX NSDI 2012*.

[8] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed Data Structures over a Shared Log. In *ACM SOSP 2013*.

[9] BERNSTEIN, P. A., AND DAS, S. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Eng. Bull. 38*, 1 (2015), 32–49.

[10] BERNSTEIN, P. A., DAS, S., DING, B., AND PILMAN, M. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *ACM SIGMOD 2015*.

[11] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR 2011*.

[12] BEVILACQUA-LINN, M., BYRON, M., CLINE, P., MOORE, J., AND MUIR, S. Sirius: Distributing and Coordinating Application Reference Data. In *USENIX ATC 2014*.

[13] BIRMAN, K. P., AND JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS) 5*, 1 (1987), 47–76.

[14] BREWER, E. A. Towards robust distributed systems. In *PODC 2000*.

[15] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. Tardis: A branch-and-merge approach to weak consistency. In *ACM SIGMOD 2016*.

[16] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR) 36*, 4 (2004), 372–421.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP 2015*.

[18] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[19] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. In *VLDB 2015*.

[20] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM SOSP 1989*.

[21] GRAY, J. N. Notes on data base operating systems. In *Operating Systems*. Springer, 1978, pp. 393–481.

[22] GUERRAOUI, R., AND SCHIPER, A. Total order multicast to multiple groups. In *IEEE ICDCS 1997*.

[23] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX OSDI 2016*.

[25] KAMINSKY, A. K. M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *USENIX ATC 2016*.

[26] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., DAHLIN, M., ET AL. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *USENIX OSDI 2012*.

[27] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS) 10*, 4 (1992), 360–391.

[28] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[29] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[30] LAMPORT, L. Generalized consensus and paxos. Tech. rep., Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[31] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *ACM SOSP 2015*.

[32] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N. M., AND RODRIGUES, R. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX OSDI 2012*.

[33] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI 2016*.

[34] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. Providing persistent objects in distributed systems. In *ECOOP 1999*.

[35] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *USENIX NSDI 2013*.

[36] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *USENIX NSDI 2013*.

[37] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *ACM SOSP 2013*.

[38] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended virtual synchrony. In *IEEE ICDCS 1994*.

[39] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distributed Transactions. In *USENIX OSDI 2014*.

[40] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *USENIX OSDI 2016*.

[41] NAWAB, F., ARORA, V., AGRAWAL, D., AND EL ABBADI, A. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. In *EDBT* (2015), pp. 13–24.

[42] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *ACM PODC 1988*.

[43] ONGARO, D., AND OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In *USENIX ATC 2014*.

[44] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *ACM SOSP 1997*.

[45] RODRIGUES, L., GUERRAOUI, R., AND SCHIPER, A. Scalable atomic multicast. In *IEEE ICCCN 1998*.

[46] SCHNEIDER, F. B. The state machine approach: A tutorial. In *Fault-tolerant distributed computing* (1990), Springer, pp. 18–41.

[47] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[48] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *ACM SOSP 2011*.

[49] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM SOSP 1995*.

[50] THOMSON, A., AND ABADI, D. J. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *USENIX FAST 2015*.

[51] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *ACM SIGMOD 2012*.

[52] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos made moderately complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 42.

[53] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A flexible group communication system. *Communications of the ACM 39*, 4 (1996), 76–83.

[54] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *USENIX OSDI 2004*.

[55] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *ACM SIGMOD 2017*.

[56] WEI, M., ROSSBACH, C., ABRAHAM, I., WIEDER, U., SWANSON, S., MALKHI, D., AND TAI, A. Silver: a scalable, distributed, multi-versioning, always growing (Ag) file system. In *USENIX HotStorage 2016*.

[57] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., ET AL. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *In USENIX NSDI 2017*.

[58] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining ACID and BASE in a Distributed Database. In *USENIX OSDI 2014*.

[59] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via modular concurrency control. In *ACM SOSP 2015*.

[60] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. Building consistent transactions with inconsistent replication. In *ACM SOSP 2015*.

[61] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP 2013*.

Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently

Kaushik Veeraraghavan Justin Meza Scott Michelson Sankaralingam Panneerselvam
Alex Gyori David Chou Sonia Margulis Daniel Obenshain Shruti Padmanabha
Ashish Shah Yee Jiun Song Tianyin Xu*

{kaushikv, jjm, sdmich, sankarp, gyori, davidchou, frumious, danielo, shrupad, ahish9, yj, tianyin}@fb.com

Facebook Inc. *UIUC

Abstract

We present Maelstrom, a new system for mitigating and recovering from datacenter-level disasters. Maelstrom provides a traffic management framework with modular, reusable primitives that can be composed to *safely* and *efficiently* drain the traffic of interdependent services from one or more failing datacenters to the healthy ones.

Maelstrom ensures safety by encoding inter-service dependencies and resource constraints. Maelstrom uses health monitoring to implement feedback control so that all specified constraints are satisfied by the traffic drains and recovery procedures executed during disaster mitigation. Maelstrom exploits parallelism to drain and restore independent traffic sources efficiently.

We verify the correctness of Maelstrom's disaster mitigation and recovery procedures by running large-scale tests that drain production traffic from entire datacenters and then retore the traffic back to the datacenters. These tests (termed drain tests) help us gain a deep understanding of our complex systems, and provide a venue for continually improving the reliability of our infrastructure.

Maelstrom has been in production at Facebook for more than four years, and has been successfully used to mitigate and recover from 100+ datacenter outages.

1 Introduction

Modern Internet services are composed of hundreds of interdependent systems spanning dozens of geo-distributed datacenters [7, 20]. At this scale, seemingly rare natural disasters, such as hurricanes blowing down power lines and flooding [29, 42], occur regularly. Further, man-made incidents such as network fibercuts, software bugs and misconfiguration can also affect entire datacenters [22, 33, 37]. In our experience, outages that affect one or more datacenters cannot be addressed by traditional fault-tolerance mechanisms designed for individual machine failures and network faults as co-located redundant capacity is also likely impaired.

In a disaster scenario, *mitigation* is almost always the first response to reduce user-visible impact, before root causes are discerned and systems are recovered. In our experience, outages affecting physical infrastructure take a long time to repair as they often involve work by on-site maintenance personnel. Software failures can be hard to debug and fix, and thus it is hard to guarantee a resoluton time [9, 22, 23, 55].

The basic idea of disaster mitigation is to quickly *drain traffic*—redirect requests originally sent to failing datacenters and reroute them to healthy datacenters. Our assumption when draining traffic is that a disaster affects only a fraction of our overall infrastructure—this assumption is reasonable because most natural disasters (e.g., hurricanes and earthquakes) are locality based. For software failures caused by bugs and misconfiguration, we adopt a locality-based staged rollout strategy, partially driven by our ability to use Maelstrom to quickly drain traffic from an affected datacenter.

We find that most failures are not instantaneous and thus can be detected and mitigated in time. For instance, we had about one week of notice before Hurricane Florence made landfall in North Carolina on September 15, 2018. This advance notice allowed us to plan and execute mitigations were Facebook's Forest City datacenter to be affected. Further, it is far more likely that a failure affects parts of a datacenter or certain infrastructure components (e.g., several network backbone cables) than resulting in total loss of a physical datacenter. In all these cases, developing the mechanism to quickly redirect user traffic as well as inter-service traffic, which we term "draining traffic," is key to disaster readiness.

The conceptually simple idea of draining traffic turns out to be rather challenging in practice. In our experience, disasters often trigger failures that affect multiple interdependent systems simultaneously. Ideally, every system should be implemented with a multi-homed design [25], where any traffic can be sent to and served by any datacenter. However, we observe that most of today's Internet services are composed of a number of heterogeneous systems including singly-homed and failover-based systems with complex, subtle dependen-

cies, and distinct traffic characteristics [12, 29, 42, 48].

The most challenging aspect of mitigation is to ensure that dependencies among systems are not violated. For instance, in a distributed caching system, if we drain cache invalidation traffic before redirecting read traffic from clients, we risk serving stale data. Or, in a web service, if we drain intra-datacenter traffic between web servers and backend systems before redirecting user requests, we risk increasing response latency due to cross-datacenter requests. Hence, we need a disaster mitigation system that can track dependencies among services, and also sequence operations in the right order.

Different systems may require customized mitigation procedures due to their distinct traffic characteristics, e.g., draining stateless web traffic requires a different procedure from draining stateful database traffic. Without unified, holistic tooling, each system might end up maintaining their own, incompatible disaster mitigation scripts that cannot be composed or tuned for scenarios with varying levels of urgency. As shown in §5.3, draining systems sequentially can significantly slow down the mitigation process, and prolong the impact of a disaster.

Disaster mitigation and recovery strategies also need to monitor shared resources, such as network bandwidth and datacenter capacity. Naïvely redirecting all traffic from one datacenter to another could overwhelm the network and trigger cascading failures.

1.1 Maelstrom for Disaster Mitigation & Recovery

We present Maelstrom, a system used for mitigating and recovering from datacenter-level disasters[1] at Facebook. Maelstrom *safely* and *efficiently* drains traffic of interdependent systems from one or more failing datacenters to the healthy ones to maintain availability during a disaster. Once the disaster is resolved, Maelstrom restores the datacenter to a healthy state.

Maelstrom offers a generic traffic management framework with modularized, reusable primitives (e.g., shifting traffic, reallocating containers, changing configuration, and moving data shards). Disaster mitigation and recovery procedures are implemented by customizing and composing these primitives. Inter-system dependencies specify the order of executing the primitives, and resource constraints control the pace of executing individual primitives. This design is driven by two observations: 1) while each system has its own procedures for mitigation and recovery, these procedures share a common set of primitives, and 2) different procedures share similar high-level flows—draining traffic while maintaining system health and SLAs. Therefore, it is feasible to build a generic system to satisfy the needs of a wide variety of systems with heterogeneous traffic characteristics.

To ensure safety, Maelstrom coordinates large-scale traffic shifts by respecting inter-system dependencies and resource constraints. Dependencies are rigorously encoded and maintained. We employ critical path analysis to identify bottlenecks and decrease time to mitigate disasters. Maelstrom implements a closed feedback loop to drain traffic as fast as possible without compromising system health. In order to mitigate disasters efficiently, Maelstrom exploits parallelism to drain independent traffic sources, which significantly speeds up execution of the mitigation and recovery procedures.

We find that Maelstrom makes disaster mitigation and recovery significantly easier to understand and reason about, in comparison to monolithic, opaque scripts. Maelstrom also incorporates extensive UI support to display the mitigation and recovery steps, and their runtime execution states, to assist human proctoring and intervention in disaster scenarios (cf. §3).

1.2 Drain Tests for Verifying Disaster Readiness

We employ Maelstrom to run different types of large-scale tests that simulate real-world disasters. We find that annual, multi-day failure drills such as DiRT [29] and GameDay [42] are useful to verify that entire datacenters can be shutdown and restarted. However, besides these annual tests, we desire a regimen of continuous tests that can be executed at daily and weekly frequencies to ensure that our mitigation and recovery keep up with rapidly-changing systems and infrastructure.

We present our practical approach, termed *drain tests*, to address the challenge. A drain test is a fully automated test that uses Maelstrom to drain user-facing and internal traffic from our datacenters in the same way as if these datacenters are failing. Running drain tests on a regular basis enables our systems to always be prepared for various disaster scenarios by maintaining and exercising the corresponding mitigation procedures. Drain tests also force us to gain a deep understanding of our complex, dynamic systems and infrastructure, and help us plan capacity for projected demand, audit utilization of shared resources, and discover dependencies.

Drain tests operate on live production traffic and thus could be disruptive to user-facing services, if not done carefully. It has taken us multiple years to reach our current state of safety and efficiency. Our original tests only targeted one stateless system: our web servers. The first set of drain tests were painful—they took more than 10 hours, and experienced numerous interruptions as we uncovered dependencies or triggered failures that resulted in service-level issues. As we built Maelstrom and began using it to track dependencies, drain tests gradually became smooth and efficient. After a year, we extended

[1] Maelstrom does not target machine-level failures (which should be tolerated by any large-scale system), or software bugs and misconfiguration that can be immediately reverted.

drain tests to two more services: a photo sharing service and a real-time messaging service. Currently, Maelstrom drains hundreds of services in a fully automated manner, with new systems being onboarded regularly. We can drain all user-facing traffic, across multiple product families, from any datacenter in less than 40 minutes.

1.3 Contributions

Maelstrom has been in operation at Facebook in the past 4 years, and has been used to run hundreds of drain tests and has helped mitigate 100+ disasters. The paper makes the following contributions:

- Maelstrom is the first generic framework that can drain heterogeneous traffic of interdependent systems safely and efficiently to mitigate datacenter-level disasters.
- We introduce drain tests as a novel reliability engineering practice for continuously testing and verifying the disaster readiness of Internet services.
- We share the lessons and experiences in running regular drain tests, as well as mitigating real disasters at a large-scale Internet service.

2 Background

This section provides an overview of Facebook's infrastructure and traffic management primitives which are similar to other major Internet services [10, 20, 32].

2.1 Infrastructure Overview

As Figure 1 shows, user requests to `www.Facebook.com` are sent to an ISP which maps the URL to an IP address using a DNS resolver. This IP address points to one of the tens of edge locations (also known as Point-of-Presence or PoPs) distributed worldwide. A PoP consists of a small number of servers, typically co-located with a peering network [43, 54]. A PoP server terminates the user's SSL session and then forwards the request on to an L4 load balancer (Edge LB) which forwards the request on to a particular datacenter. A user request can be served from any of our datacenters.

We group machines in a datacenter into logical *clusters* such as frontend clusters composed of web servers, backend clusters of storage systems, and generic "service" clusters. We define a *service* as the set of subsystems that support a particular product.

Within a datacenter, an L7 web load balancer (Web LB) forwards the user request to a web server in a frontend cluster. This web server may communicate with tens or hundreds of services, and these services typically need to further communicate with other services and backends, to gather the data needed to generate a response. We employ a set of service load balancers (Service LBs) to distribute requests amongst service and backend clusters. The web server handling the user request is also responsible for returning the response to the PoP which then forwards it on to the end user.

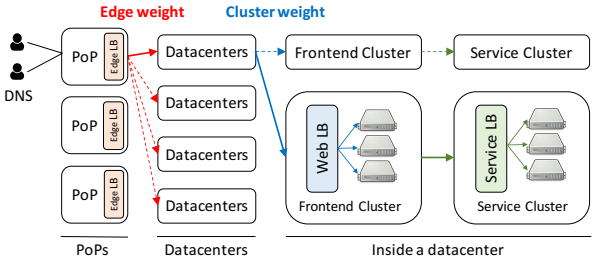


Figure 1: An overview of Facebook's infrastructure. The configurable edge and cluster weights determine how user requests are routed from PoPs to particular datacenters, and then on to particular clusters.

Traffic	Affinity	State	Strategy
Stateless	—	—	reroute
Sticky	✓	—	reroute → tear down
Replication	—	✓	customized
Stateful	✓	✓	master promotion

Table 1: Traffic type, property, and mitigation strategy (cf. §2.3).

2.2 Traffic Management Primitives

The PoP server parses each request URI and maps it to a service. Our traffic management system assigns each service a virtual IP (VIP). Traffic for each VIP is controlled by two configurable values: *edge weight* and *cluster weight*. Edge weights specify the fraction of requests that the PoP should forward to each of the datacenters. Cluster weights specify the fraction of requests that each cluster is capable of handling.

Since PoPs and frontend clusters are stateless, a user request can be sent to any PoP and forwarded to any frontend web server. This property allows us to programmatically reconfigure edge and cluster weights to reroute traffic in disaster scenarios. For instance, if a network fiber-cut disconnects a datacenter from the rest, we push out a configuration change to all PoPs that sets the edge weight for the disconnected datacenter to 0; this results in the traffic originally sent to the failing datacenter being routed to the other datacenters.

Internal service traffic (e.g., RPC traffic) within and across datacenters are controlled by L7 service load balancers based on configurable knobs in a similar vein.

2.3 Traffic Types

Table 1 categorizes the traffic types of different systems based on affinity and state properties, as well as the common strategies for draining them during disasters.

- *Stateless.* The vast majority of web traffic is stateless, consisting of users' web requests directed from PoPs

to one or more datacenters. Stateless traffic can be drained by rerouting it away from a failing datacenter, or from particular sets of clusters, racks, or machines.

- *Sticky.* Interactive services (e.g., messaging) improve user experience by pinning requests to particular machines that maintain the state for a user in a session. Sticky traffic can be drained by rerouting incoming session requests and tearing down the established sessions to force them reconnect to other machines.
- *Replication.* In a disaster, we may need to alter or even stop replication traffic from egressing or ingressing the failing datacenter for distributed storage systems. The replicas can be re-created in other datacenters to serve reads. This requires configuration changes or other heavyweight changes that influence resource sharing, such as intra- and inter-datacenter networks.
- *Stateful.* For master-slave replication based systems, the mitigation for a master failure is to promote a secondary to be the new master. This may require copying states from the failing datacenter to the new. The state copy requires careful control based on the network capacity to transfer data out to healthy machines.

3 Maelstrom Overview

Maelstrom is a disaster mitigation and recovery system. During a datacenter-level disaster, operators[2] use Maelstrom to execute a *runbook* that specifies the concrete procedure for mitigating the particular disaster scenario by draining traffic out of the datacenter; after the root causes are resolved, a corresponding recovery runbook is used to restore the traffic back.

Maelstrom provides a generic traffic management framework. A runbook can be created via Maelstrom's UI by composing a set of *tasks*. A task is a specific operation, such as shifting a portion of traffic, migrating data shards, restarting container jobs, and changing configuration. Tasks can have *dependencies* that determine the order of execution—a task should not be started before its dependent tasks are completed. Figure 2 shows an example of a runbook and its corresponding tasks. We elaborate the runbook-based framework in §4.2.

Every service maintains its own *service-specific runbooks* for disaster mitigation. Taking our interactive messaging service as an example, the runbook for draining the service's sticky traffic (upon software failures in a datacenter) includes two tasks in order: 1) redirecting new session requests to the other datacenters, and 2) terminating established sessions in the failing datacenter to force them reconnect. A recovery runbook can be used to restore messaging traffic back to the datacenter.

[2]In this paper, we use "operators" as a general term for anyone helping with operations, including Software Engineers, Site Reliability Engineers, Production Engineers, and System Administrators.

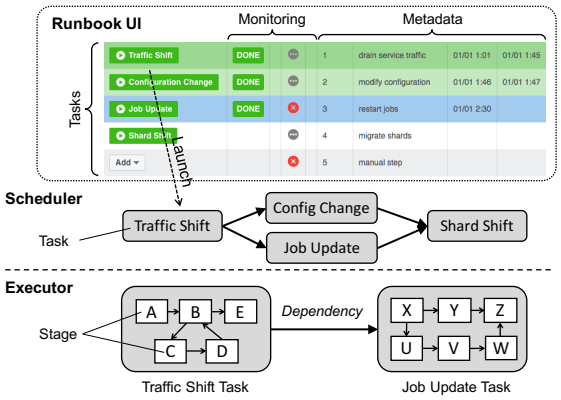


Figure 2: Maelstrom executes *runbooks,* each specifying the procedure for mitigating a particular disaster scenario. A runbook is composed of interdependent *tasks* (e.g., traffic shift and job updates). These tasks are scheduled by Maelstrom's *scheduler* based on their dependencies, and are executed in multiple stages by Maelstrom's *executor*. Maelstrom monitors and displays the runtime status of tasks in its UI.

If an entire datacenter is down (e.g., due to network fibercuts that disconnect it from our infrastructure), a *datacenter evacuation runbook* will be used to drain traffic of all the services in the datacenter. A datacenter evacuation runbook is composed of service-specific runbooks, where each runbook drains or restores the traffic for a particular service deployed in a datacenter. These service-specific runbooks are aggregated through external dependencies that link tasks in different runbooks.

Runbooks are executed by Maelstrom's runtime engine consisting of two main components: 1) the *scheduler* that schedules tasks to execute based on the policy specified in the runbook (including dependencies and conditions), and 2) the *executor* that is responsible for executing each individual task. A task can be executed in multiple stages based on its implementation (cf. §4.3).

As shown in Figure 2, Maelstrom is equipped with a UI that visualizes the runtime information of a runbook, including the state of every task, their dependencies, and the associated health metrics. We keep improving the UI with an operator-centric methodology. Each disaster is a learning opportunity for us to interact with operators and to improve usability. Our UI design focuses on helping operators understand the mitigation and recovery status, and on efficiently controlling the runbook execution.

At Facebook, we use Maelstrom to run different types of tests with different frequencies. Besides weekly drain tests, we also run *storm tests* at a quarterly cadence. The primary difference between a *drain test* and a *storm test* is that a drain test is focused on draining user traffic out of a datacenter as fast as possible without user perceivable impact. In contrast, a storm test extends beyond user traffic to drain all RPC traffic amongst services, stops data replication, and applies network ACLs to isolate the tested data center. Thus, a storm test is a more rigorous

endeavor that verifies that all of Facebook's products and systems can function correctly despite the total loss of a datacenter. From our understanding, storm tests are akin to Google's DiRT [29] and Amazon's GameDay [42] exercises. In this paper, we focus on drain tests as a new type of large-scale, fully-automated test for production services, which can be run on a daily or weekly basis.

3.1 Drain Tests

Maelstrom requires runbooks to always keep updated with our rapidly-evolving software systems and physical infrastructure. However, maintaining up-to-date information (e.g., service dependencies) is challenging due to the complexity and dynamics of systems at scale, akin to the observations of other cloud-scale systems [5, 29, 33].

Drain tests are our practical solution to continuously verify and build trust in the runbooks. A drain test is a *fully automated* test that uses Maelstrom to drain user-facing and internal service traffic from our datacenters in the same way as if these datacenters are failing. Internal services include various asynchronous jobs, data processing pipelines, machine learning systems, software development tools, and many other services that are key components of our infrastructure.

We run multiple drain tests per week to simulate various types of disaster scenarios (e.g., those listed in [19]) on a least-recently-tested datacenter. Tests are scheduled at different time of a day to cover various traffic patterns (e.g., peak and off-peak time). We also vary the duration of each test to understand how the rest of our infrastructure serve user and service traffic when the disaster is in effect. Running drain tests brings many benefits:

- verifying that runbooks can effectively mitigate and recover from various types of disasters and meet our recovery objectives;
- aid planning by identifying capacity needs during various disaster scenarios;
- testing the pace at which a service can offload traffic without overwhelming its downstream systems;
- auditing how shared resources are utilized to identify resource bottlenecks;
- tease apart complex inter-system dependencies and continuously discover new dependencies.

A drain test is not expected to have any user-visible or service-level impact. If this expectation is not met, we follow up with the engineering teams to understand why a given disaster scenario was not handled well, and schedule followup tests to verify fixes.

3.2 Failure Mitigation

Maelstrom was initially built for mitigating disasters of physical infrastructure. We experience a handful of incidents each year that result in the temporary catastrophic loss of one or more datacenters, usually due to power or network outages. We mitigate and recover from these disasters using datacenter evacuation runbooks.

Over time, our practice of rigorously verifying runbooks via drain tests has resulted in its evolution as a trusted tool for handling a wide variety of failures, including service-level incidents caused by software errors including bugs and misconfiguration. These service-level incidents are an order of magnitude more frequent. Note that most service incidents are recovered by reverting the buggy code or configuration changes, so traffic drains are rare. We will discuss how Maelstrom is used to deal with various failure scenarios in §5.1.

The actual failures and disasters are mitigated using the same runbooks as drain tests. Drain tests are fully automated—operators are only paged when the test triggers unexpected issues. During a disaster, operators may choose to accelerate steps to speed up mitigation.

4 Design and Implementation

4.1 Design Principles

Composability. In our experience, despite the heterogeneity of mitigation and recovery procedures of different systems, they share common structures and can be composed of a common set of primitives. Maelstrom enables services to implement their own runbooks by composing various primitives. Composability offers a number of benefits: 1) it allows Maelstrom to exploit parallelism among primitives; 2) it enforces modularity and reusability of mitigation- and recovery-related code, and 3) it makes runbooks easy to understand and maintain.

Separation of policy and mechanism. Maelstrom separates *policies* that define how traffic should be drained and restored in a specific disaster scenario and the *mechanisms* for executing traffic shifts and other related operations (cf. §2.2).

Safety as a constraint. Disaster mitigation and recovery themselves must not create new outages—when shifting traffic, Maelstrom should avoid cascading failures that overload the remaining healthy datacenters. Further, drain tests as a regular operation should not have any user-visible, service-level impact.

Embracing human intervention. We have learned that it is critical for a disaster mitigation and recovery system to embrace human intervention, even with fully automated runbooks (cf. §6). The design should minimize tedious operations to let operators focus on critical decision making.

4.2 Runbook Framework

A runbook is created through the Maelstrom UI by specifying the following information:

Task Template	Parameter	Description
`TrafficShift`	{`vip_type`, `target`, `ratio`, ...}	Shift traffic into or out of a cluster or a datacenter (specified by `target`): `vip_type` specifies the traffic; `ratio` specifies the amount of traffic to shift.
`ShardShift`	{`service_id`, `target`, `ratio`, ...}	Move persistent data shards into or out of a cluster or a datacenter via our shard manager (`target` and `ratio` have same semantics as in `TrafficShift`).
`JobUpdate`	{`operation`, `job_ids`, ...}	Stop or restart jobs running in containers.
`ConfigChange`	{`path`, `rev_id`, `content`, ...}	Revert and/or update configuration in our distributed configuration store.

Table 2: Several common templates used to materialize tasks in runbooks, and their descriptions. Note that we have omitted the optional parameters that provide fine-grained control (e.g., latency and stability optimization) from this table.

- *Task specifications.* Tasks are materialized by applying parameters to a library of templates. Table 2 lists several task templates and their descriptions.
- *Dependency graph.* We use a directed acyclic graph (DAG) to represent dependencies amongst the tasks in a runbook. Every node T_i in the DAG refers to a task in the runbook. A directed edge $T_1 \rightarrow T_2$ represents a *dependency*: Task T_1 *must precede* Task T_2, which means that T_2 can only be scheduled for execution after T_1 is completed.
- *Conditions.* A task can have pre-conditions (checking if it is safe to start) and post-conditions (determining if it is completed successfully). Pre-conditions are typically used as safeguards to ensure that the service is in a healthy state, while a post-condition could check if the target traffic reaches zero.
- *Health metrics.* Each task is associated with a number of service-health metrics, which are visualized in Maelstrom's UI to help human operators monitor the status of task execution.

Each service maintains its *service-specific runbook* for disaster mitigation and recovery. We also maintain an *evacuation runbook* for each of our datacemters which aggregates service-specific runbooks. The aggregation is accomplished by adding dependencies between tasks from different service-specific runbooks. We run the evacuation runbooks during each drain test and thus exercise all the related service-specific runbooks. Therefore, every drain test covers hundreds of services—we run tests far more often than we experience real failures.

4.3 Runtime Engine

Maelstrom's runtime engine is responsible for executing a runbook. The runtime engine consists of two components: 1) a *scheduler* that determines the order of executing tasks by tracking their dependencies, and 2) an *executor* that executes each task and validates the results.

- *Scheduler.* The scheduler generates an *optimal* schedule of task execution by parallelizing independent tasks. The scheduler marks a task `ready` for execution and sends its specification to the executor, when and only when all the parent tasks that must precede this task are `completed` and all the pre-conditions are satisfied. Note that this schedule is generated dynamically based on the runtime status of each task, and it supports operator intervention (e.g., skipping and stopping tasks).
- *Executor.* The executor materializes each task based on the parameters in the specification sent by the scheduler, and then executes the task. A task is executed in multiple *steps*. For example, a `TrafficShift` task for draining 100% web traffic out of a datacenter can be done in one step, or in five steps (with wait time in between)—each one draining 20%—based on the desired pacing (cf. §4.7).

Maelstrom models a task as a nondeterministic finite state machine, with a set of *stages* $\mathbb{S}$, a set of runtime inputs, *transitions* between stages, an initial stage $I \in \mathbb{S}$, and a set of exit stages $\mathbb{E} \subseteq \mathbb{S}$. A stage accepts one or more inputs, performs the desired action, and then optionally invokes a *transition* to the next stage. A stage can have multiple outgoing and incoming transitions. Each stage can generate outputs, as inputs for other stages. The executor starts from I and continues executing subsequent stages following the transitions, until reaching an exit stage. This design allows us to reuse stages as the basic unit for implementing task templates.

We implement a library of stages that capture common operations like instructing load balancers to alter traffic allocation [6, 17, 36, 46], managing containers and jobs [12, 44, 51], changing system configuration [45, 47], migrating data shards [1, 13], etc. We also implement various helper stages for communication and coordination, such as `Barrier`, `TimedWait`, and `ChangeLog`.

4.4 Executing Runbooks: Putting It All Together

Figure 3 illustrates how Maelstrom executes a service-specific runbook to drain traffic of a messaging service. Maelstrom executes two tasks in order: 1) redirecting new incoming session requests away, and 2) tearing down the remaining established sessions so clients can reconnect to machines in other datacenters. Maelstrom verifies that all of a task's parent dependencies are drained, and pre-conditions are satisfied. The sec-

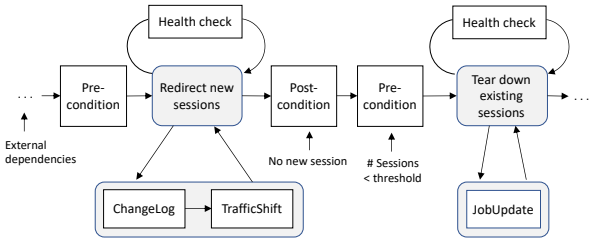


Figure 3: A runbook to drain a messaging service's sticky traffic. The runbook has two tasks: redirecting new sessions and tearing down existing sessions—both are executed in multiple steps.

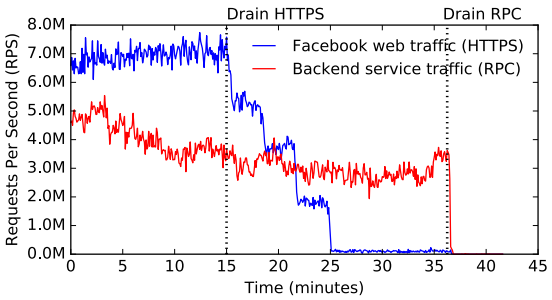


Figure 4: An example of a service dependency that determines the order of drains. A web service with HTTPS traffic communicates with backend services via RPC traffic (we say the web service *depends on* the backend services). So, the web service's HTTPS traffic must be drained *before* the backend service's RPC traffic is drained.

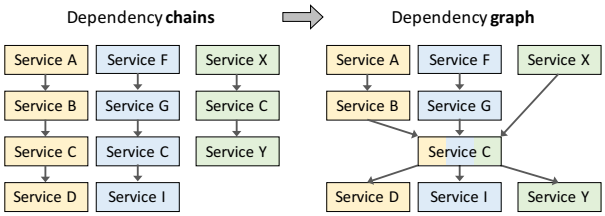


Figure 5: To build a runbook, we start with independent dependency *chains* (left). We identify *highly connected components (HCCs)*, like Service C, and merge the dependency chains at the HCCs to form a dependency *graph* (right).

ond task uses its pre-condition as a safety check to confirm that the number of active sessions has dropped below a preset threshold to minimize the impact of tearing down all existing sessions. Maelstrom marks a task as `completed` when the post-conditions are met.

Drains are blocked if the pre-/post-conditions are not met, because this signifies that the service is in an unexpected state. Maelstrom compares the duration of each task to the 75th percentile of the execution time of prior tests/incidents to determine whether the task is stuck in execution. If so, the operator will be alerted. We prioritize safety over speed, and stall subsequent operations until an operator intervenes—we find stalls to be a rare event that occurs only once every several dozen tests. When handling actual failures, Maelstrom allows human operators to override particular pre-/post-conditions if they wish—each of these overrides are logged and reviewed in postmortems to improve automation.

Maelstrom's traffic drains and restorations are guided by a variety of constraints:

- *physical and external constraints*, including network over-subscription within a datacenter, cache hit rate, I/O saturation in backend systems, etc.
- *service-specific constraints*—different types of traffic have distinct constraints, e.g., draining sticky traffic is prone to a thundering-herd effect as session establishment is resource intensive; draining stateful traffic leads to master promotion which requires the slaves to catch up with all updates, or restore states from logs; restoring replication traffic requires syncing updates with volumes proportional to down time, and the sync speed is constrained by network bandwidth.

4.5 Dependency Management

Broadly, we find that there are three common relationships amongst services that manifest as dependencies:

- *Bootstrapping*: A service depends on low-level system components to prepare the execution environment and setup configuration before serving traffic.
- *RPC*: A service makes remote procedure calls (RPCs) to fetch data from other services.
- *State*: Traffic can have states. For instance, a service with sticky traffic may depend on a proxy to coordinate and manage session establishment.

Discovery and sequencing. We employ a combination of methods to discover the aforementioned dependencies when onboarding a new service to Maelstrom. First, we design scenario-driven questionnaires to help service teams reason about their dependencies with upstream and downstream services under different types of failures and disasters. Moreover, we leverage our tracing systems (e.g., [26]) to analyze how the target service interacts with other services through RPC and network communications. Our analysis incorporates our service discovery systems to map the interactions to specific services. We further analyze the traces and logs of software components to reason about state-based dependencies based on the causal models of service behavior [16]. Figure 4 illustrates the dependency between traffic of two services, which enforces the order of drains.

After identifying dependencies, the next challenge is to sequence drains amongst multiple interdependent systems in the right order. We tackle this problem by first organizing services into chains of parent-child dependencies in a disconnected dependency graph (we often do not have one single complete graph at a time). Next, we identify common services across chains—the common

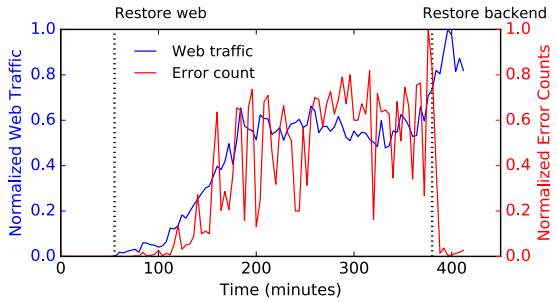


Figure 6: Restoring web traffic (left y-axis) at minute 55 caused a proportional increase in errors (right y-axis) until a backend service that the web traffic depended on was restored at minute 375.

services are often highly connected components (HCCs) that hold the dependency graph, as illustrated in Figure 5. Draining a HCC service will likely require us to drain its parents first; once the HCC service is drained, its children can likely be drained concurrently.

Continuous verification. We use Maelstrom to empirically verify that independent services can be drained and restored in parallel. For a new service, we use Maelstrom to cautiously validate the specified dependencies via small-scale drain tests, while closely monitoring the health of all involved services. We gradually enlarge the radius of the drain tests until all levels of traffic drains can be performed regularly. We find this process to be time consuming but worthwhile, as a principled way of verifying dependencies in a controlled, scalable manner.

Figure 6 illustrates how a previously unknown dependency was discerned in a drain test. This issue was caused by an out-of-order restore, where a backend service was drained while its dependent web traffic was restored. This made the error count proportional to the web traffic, as the web service was trying to query the unavailable backend. The error rate went down to zero after the operator also restored the backend service. After this test, the dependency was added into the runbook, together with improved monitoring of the error count.

Critical path analysis. Maelstrom performs automated critical path analysis after each drain test as well as each disaster mitigation and recovery event. Critical path analysis helps us optimize mitigation time by identifying bottlenecks in our dependency graphs.

When adding a new dependency into an existing runbook, we run drain tests to check if the new dependency is on the critical path or not. If it is, we engage with the service team responsible for that dependency to optimize drain and recovery performance. We also actively examine dependencies on slow, heavy tasks (e.g., data shard migration) to try to move these dependencies off the critical path. If a dependency lengthens the critical path, the service team evaluates whether the dependency provides value, given its failure mitigation cost.

Maelstrom allows a dependency to be tagged as *weak*, while by default all dependencies are *strong*. Strong dependencies affect correctness, and thus are expected to be respected in most failure scenarios. Weak dependencies affect a system's performance and reliability SLA. Drain tests respect both strong and weak dependencies. During a disaster, operators can override weak dependencies to stabilize a system or speed up mitigation. For instance, in case of a fibercut disconnecting a datacenter, an operator might move all user traffic to a different datacenter in a single step which minimizes user impact, but might affect the hit rate of any underlying caching systems, and possibly push backend storage systems to their limits. We curate weak dependencies by analyzing the dependencies on the critical path as discussed above. We also perform experiments that intentionally break weak dependencies in a controlled fashion to assess the corresponding service-level impact.

4.6 Preventing Resource Contention

Safely draining traffic with Maelstrom involves ensuring that shared resources (e.g., server compute capacity and network bandwidth) do not become overloaded during a drain. Our approach to reducing the effect of resource contention is guided by the following three principals:

- *Verifying capacity.* We verify that the shared infrastructure has enough capacity to absorb the spikes in utilization caused by draining with Maelstrom through regular testing. Since Facebook has full monitoring and control of its backbone network, we can observe how draining affects peak network utilization. When bottlenecks arise during tests, we work with teams to update our routing policies, traffic tagging and prioritization schemes, or bandwidth reservation configuration so we can drain services safely. At the network level, we provision multiple diverse paths both intra- and inter-datacenters, and plan 75% utilization for our switches [34].
- *Prioritizing important traffic.* To handle the event of a widespread failure where shared resources cannot support demand, we have a prioritization scheme for how we drain traffic from a datacenter. We prioritize draining user-facing traffic as soon as possible to limit the user-perceivable impact of a failure, and then drain stateful service traffic. This ensures that the effect of the drain on an end user is minimized, and it also minimizes the overhead of state migration.
- *Graceful degradation.* Finally, we plan for systems to degrade gracefully in the case of resource overload. Some systems employ PID controllers to reduce the complexity of serving requests (e.g., by incrementally turning off ranking algorithm complexity to reduce server compute capacity). Other systems are able to

respond automatically to resource contention by performing large-scale traffic redirection, while safely accounting for the effect of traffic changes.

4.7 Pacing and Feedback Control

Maelstrom implements a closed feedback loop to pace the speed of traffic drains based on extensive health monitoring. The drain pace is determined by the `step_size` (traffic fraction to reduce) and `wait_time` before the next step. The runbook uses past drain parameters from tests/incidents as a starting value for step size and wait time. When running an actual drain, these parameters are further tuned to be more aggressive or conservative based on the health of underlying systems.

Our pacing mechanism seeks to balance safety and efficiency—we wish to drain as fast as possible without overloading other datacenters. Specifically, Maelstrom breaks down a drain operation into multiple steps, and for each step, tunes the weights such that no traffic shift breaches the health metrics of any datacenter. For instance, when draining web traffic from 100% to 0%, Maelstrom typically does not drain in one step (which could have ripple effect such as significant cache misses). Instead, the drain takes multiple steps (with specified `wait_time` in between), gradually increasing the traffic shift proportion, in order to allow cache and other systems to warm up with smoothly increasing load without getting overwhelmed. The health metrics are also displayed in Maelstrom's UI, so operators can audit operations and intervene as needed.

Maelstrom reads the health data maintained as time series. In our experience, a few key metrics from each service can provide good coverage of their health, and we infrequently need to add new metrics.

We use drain tests to experiment with various starting speeds. Based on the empirical mapping from speed to health metric impact, we tune the default value to the maximal speed without compromising health or safety.

4.8 Fault Tolerance

Maelstrom is designed to be highly fault tolerant in the presence of both component and infrastructure failures. We deploy multiple Maelstrom instances in geo-distributed datacenters so at least one instance is available even when one or more datacenters fail. We also have a minimal version of Maelstrom that can be built and run on any of our engineering development servers.

We verify the correctness of runbooks by leverage continuous tests that validate the invariants in every service's runbook including checking for circular dependencies, reachability (no inexistent dependencies), duplication, ordering (every drain step is undone with a restore), and configuration (mandatory parameters are always set). If a test fails, the service's oncall engineers will be notified to review the service's tests, dependencies, health indicators, etc. If all tests pass, but other correctness violations manifest during a drain test (due to insufficient tests), the disaster-recovery team will arbitrate between services to ensure that problems are fixed and continuous tests are updated. As discussed in §4.4, live locks (e.g., due to failures of task execution or condition checks) are detected by comparing the execution time of tasks with the 75th percentile of prior running time.

Maelstrom stores its metadata and runtime state in a highly-available, multi-homed database system. Both task and stage level state is recorded so both the scheduler and executor can be recovered in case of failure. Maelstrom also records the state of each task (`waiting`, `running`, or `completed`) into the database so it can resume at the last successful step of a drain or recovery procedure. For a running task, Maelstrom records the runtime state of each stage and transition in the database based on the state machine it generated. Hence, if there is a crash of Maelstrom (including both the scheduler and the executor), we can use standard recovery techniques to read the last committed state from the database to initialize Maelstrom and resume the execution.

Maelstrom also relies on a highly-available time-series database to fetch health monitoring data [39]. The database continuously provides data even in the presence of failures by varying the resolution of data points.

5 Evaluation

Maelstrom has been in use at Facebook for more than four years, where it has been used to run hundreds of drain tests, and helped mitigate and recover from 100+ datacenter-level failures and service-level incidents.

Our evaluation answers the following questions:

- Does Maelstrom enable us to mitigate and recover from real disasters safely and efficiently?
- Does Maelstrom provide a safe and efficient methodology for regular drain tests?
- How quickly does Maelstrom drain and restore different types of traffic?

5.1 Mitigating Disasters

Power and network outages. Maelstrom is our primary tool to mitigate and recover from disasters impacting physical infrastructure whether caused by power outages or backbone network failures, resulting in the total or partial unavailability of datacenters. Taking the network as an example, a single fibercut almost never disconnects a datacenter; typically, one link is lost and network flows reroute over alternate paths. This rerouting procedure often takes tens of seconds and could impact users. On the other hand, a single-point-of-failure link, e.g., a trans-Atlantic or trans-Pacific optical cable, can

get cut [34]—such incidents are severe in both magnitude and duration-to-fix, thus requiring datacenter drains.

A recent incident caused by fibercuts led to the loss of over 85% of the capacity of the backbone network that connects a datacenter to our infrastructure. This incident was immediately detected as we experienced a dip in site egress traffic. The disaster was mitigated by the site operators using Maelstrom to drain all user and service traffic out of the datacenter in about 1.5 hours, with most user-facing traffic drained in about 17 minutes. The remaining network capacity was used to replicate data to the storage systems resident in that datacenter (which helps efficiently redirect user traffic back, once the fiber is repaired). It took several hours to repair the backbone network, at which point we used Maelstrom to restore all traffic back to the datacenter.

Note: when a datacenter is drained, users may experience higher latency, as they are redirected to a remote datacenter, or experience reconnection (only for sticky services). Draining faster could reduce the amount of time during which users experience increased latency. We are continually working to decrease dependencies and optimize constraints to enable faster drains.

Software failures. Maelstrom is also used to respond to service-level incidents caused by software errors, including bugs and misconfiguration [22, 23, 33, 52, 53]. These incidents are typically triggered in two ways:

- *software errors in ongoing rollouts.* Despite the wide adoption of end-to-end testing, canaries, and staged rollout, bugs or misconfiguration can still make their way into production systems.
- *latent software errors.* A software release or configuration change might trigger latent bugs or misconfiguration residing in production systems.

In both cases, any error or disruption ought to trigger an alert and inform operators. The operators need to decide between two options: reverting the offending change(s), or fixing forward after diagnosing the problem.

Unfortunately, neither of these options is trivial. First, it may take time to identify the offending change (or changes) due to the challenge of debugging large-scale distributed systems. Second, rollback to an early version may cause other issues such as version incompatibility, which can result in other failures. Third, it takes time to diagnose, code up a fix, test it thoroughly, and then deploy it into production [55]. During this time, the error continues to manifest in production.

Maelstrom provides a pragmatic solution for reducing the impact of failures by moving diagnosis and recovery out of the critical path—it simply drains service-specific traffic from the failing datacenters when failures are detected. We find that this mitigation approach is robust when paired with a locality-based, staged rollout strategy for all software and configuration changes.

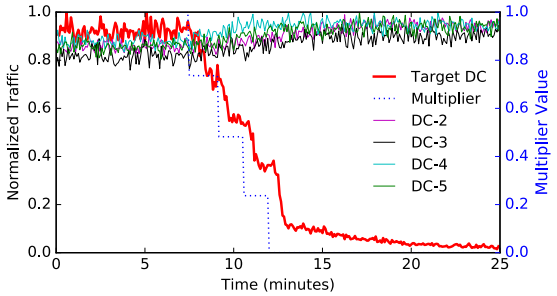


Figure 7: Draining stateless traffic. We apply a multiplier (dotted line) to the edge weight (cf. §2) of stateless traffic in Target DC to drain stateless traffic. The long tail of Target DC traffic is from DC-internal requests that are controlled separately from the edge weight.

Maelstrom was used to mitigate a recent service incident where a configuration change was rolled out to all instances of our search aggregator deployed in one of our datacenters. The configuration change inadvertently triggered a new code path and exposed a latent bug in the aggregator code (a dangling pointer). All the instances in the datacenter immediately crashed due to segfaults.

This incident was mitigated by draining service traffic from the datacenter where the misconfigured search aggregators were deployed. Detecting the incident took only 2 minutes as it immediately triggered alerts. It took 7 minutes to drain service requests out of the affected datacenter using Maelstrom. Failure diagnosis (identifying the root cause) took 20 minutes. Thus, Maelstrom reduced the duration of service-level impact by about 60%.

5.2 Draining Different Types of Traffic

5.2.1 Service-specific Traffic

Stateless traffic. Figure 7 shows how Maelstrom drains stateless web traffic of one of our services out of a target datacenter in a drain test. We normalize the data, because different datacenters have different sizes (in terms of the magnitude of traffic served) and we want to highlight the relative trends of each datacenter during the drain. The runbook for draining web traffic includes a `TrafficShift` task which manipulates the edge weights of the target datacenter by applying a *drain multiplier* between [0.0, 1.0]. The drain was executed in multiple steps indicated by the drain multiplier changes in Figure 7. Splitting the drain into multiple steps prevents traffic from shifting too fast and overloading the other datacenters (cf. §4.7).

As shown in Figure 7, the traffic follows the changes of the drain multiplier instantly. *Maelstrom can drain stateless traffic fast.* Maelstrom can drain traffic of most of our web services out of a datacenter in less than 10 minutes without any user-visible, service-level impact.

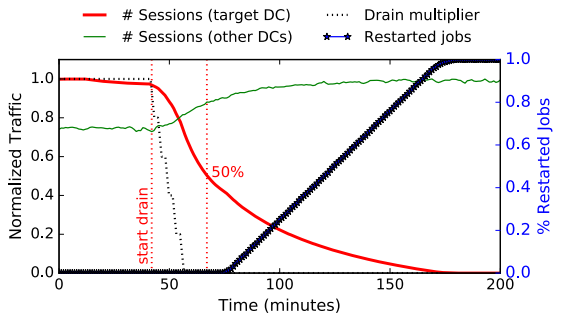


Figure 8: Draining sticky traffic. We drain sticky traffic by first applying a multiplier (dotted line) to the edge weight of a target DC (similar to stateless traffic). We then restart the jobs in the target DC to force already-established sessions in the target DC to reconnect in other DCs.

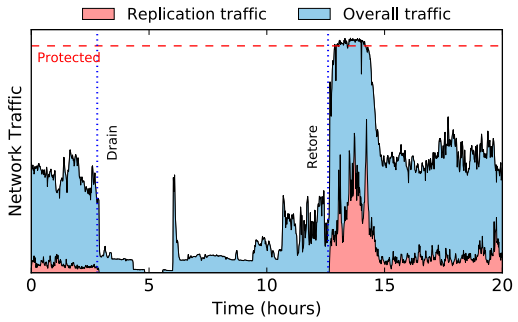


Figure 9: Drain and restore replication traffic to mitigate an incident. The "protected" line denotes the limit of steady state network utilization (which is 75% of the maximum capacity, cf. §4.6), as it includes the buffer for maintenance and upgrades. The replication service had a significant spike that, combined with network utilization by other services, consumed all available network bandwidth for recovery.

The 10-minute duration is used as a baseline for draining web traffic during real disasters (cf. §5.1).

Sticky traffic. Figure 8 shows how Maelstrom drains sticky traffic for a messaging service. This runbook contains two tasks as described in §4.4: (1) changing edge weights to redirect new, incoming session requests (at the 42nd minute), and then (2) tearing down established sessions by restarting container jobs, if the client can still connect to the datacenter (at the 75th minute). Figure 8 shows the effects of these two tasks—it took about 25 minutes to reduce the number of sessions down to 50%, and the remaining time to restart jobs and reset connections. Note that we need to pace job restarts to avoid a thundering-herd effect caused by computationally expensive session establishment. During real disasters, we find that clients' connections are severed due to network disconnections or server crashes, so drains are faster.

Replication traffic. Figure 9 shows how Maelstrom drains and restores replication traffic of a multi-tenant storage system when the target datacenter was subject to a network partition. Maelstrom drains replication traffic by disabling the replication to the storage nodes in the target datacenter after pointing read traffic away from the replicas. The drain was smooth, but the restoration caused an incident. Upon re-enabling replication for recovery, the system attempted to resolve its stale state as quickly as possible, transferring data from other datacenters at more than 10× the steady-state rate. This saturated network capacity at multiple layers in our backbone and datacenter network fabric and triggered production issues in other systems that shared the network. Figure 9 shows how a spike in network bandwidth consumption from a replication service can crowd out other services. In this incident, we found that other critical services could not transfer data cross-region which resulted in delays to serve user traffic. We have addressed this problem by incorporating network utilization limits for different services, and also considering network usage when running recovery for multiple services in parallel.

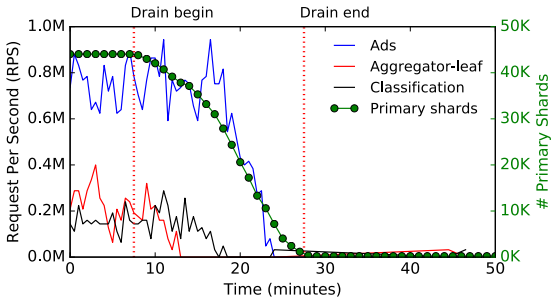


Figure 10: Draining stateful traffic. Maelstrom moves primary data shards from a storage system and simultaneously drains traffic from the services that access the storage system.

Traffic	Service	# Tasks	# Steps	Drain Time
Stateless	Web service	1	10	10 min
Sticky	Messaging	2	1 → 5	3 min → 61 min
Replication	KV store (replica)	1	1	3 min
Stateful	KV store (master)	1	24	18 min

Table 3: Time for draining different types of traffic of representative services at Facebook. The time is collected from our recent drain tests. For sticky traffic, → denotes the two tasks for draining the traffic.

Stateful traffic. Figure 10 shows how Maelstrom drained stateful traffic of three services: an "ads", an "aggregator–leaf", and a "classification" service. All three services store their data in a multi-tenant stateful storage system where the data are sharded. The storage system distributes replicas of each shard in multiple datacenters to ensure high availability. During the drain, Maelstrom promotes a replica outside of the datacenter to be the new primary shard, and then shifts traffic to it.

Figure 10 plots the fraction of primary shards in the datacenter being drained. From the perspective of the storage system, each of these services is independent of the others because their data are sharded separately. This allows Maelstrom to drain writes and promote their

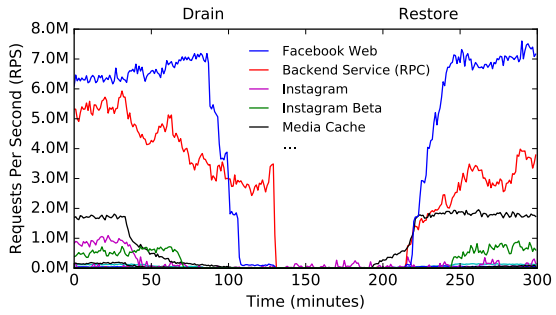


Figure 11: Draining and restoring traffic for 100+ production systems in a datacenter. Each line corresponds to the traffic of a specific service. "Facebook Web" refers to the traffic of Facebook's main web service.

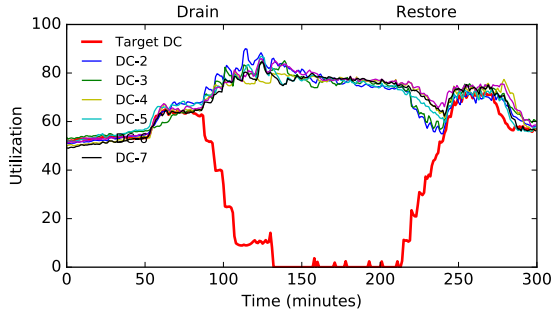


Figure 12: Datacenter utilization when the traffic of an entire datacenter is drained and restored.

masters in parallel while respecting shared resource constraints. Note that Figure 10 only highlights three services for clarity—Maelstrom drains tens of thousands of shards for hundreds of services in the datacenter.

Timing. Table 3 shows the time for Maelstrom to drain one representative service that displays each type of traffic pattern. Note that they vary significantly in duration from 3 minutes for a replication system where a drain is as simple as redirecting read requests and shutting the instance down, to a sticky service that takes 61 minutes to drain at its natural pace. Note that we encourage services to plan for different disaster scenarios but do not force particular policies for timing or reliability unless the service is in the critical path for evacuating a datacenter.

5.2.2 Draining All the Traffic of a Datacenter

Figure 11 shows a drain test that drains a wide variety of service traffic hosted in the datacenter, including both user traffic as well as internal service traffic. We see that no single service constitutes a majority of the traffic. Maelstrom achieved a high degree of parallelism while maintaining safety by ordering drains according to the dependency graph encoded in the runbook (cf. §4.5).

Figure 12 is a complement of Figure 11 that depicts the normalized utilization of both the target datacenter that is being drained of traffic, and the other datacenters that the traffic is being redirected to and then restored from. We see that once all traffic is drained, the utilization of the target datacenter drops to zero. Meanwhile, the utilization of the other datacenters increases as they need to serve more traffic. None of the remaining datacenters were overloaded—traffic is evenly distributed.

Runbook	# Tasks	# Dep.	# Tasks on CP
Mitigation (drain)	79	109	8 (9.6%)
Recovery (restore)	68	93	5 (7.4%)

Table 4: Aggregate statistics of the runbooks that drain and restore all user-facing traffic in one of our datacenters, respectively. "CP" is an abbreviation of critical path.

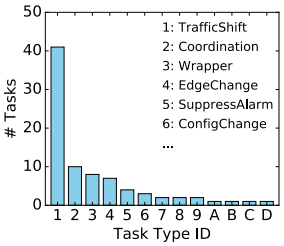


Figure 13: Histogram of number of tasks to drain and restore user-facing traffic.

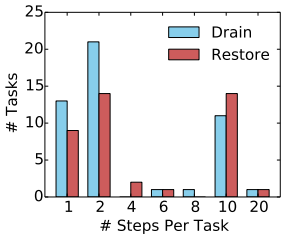


Figure 14: Histogram of number of steps per task when draining and restoring user-facing traffic.

Aggregate statistics. Table 4 analyzes the runbooks used to drain and restore all user-facing traffic and their dependent services from one of our datacenters. Our goal is to provide insights into the policies we encode in Maelstrom. The mitigation runbook consists of 79 tasks with 109 dependencies, of which less than 10% of the tasks are on the critical path. Note that this minimal critical path is not an organic outcome but rather the result of continually optimizing and pruning dependencies over four years, based on the critical path analysis described in §4.5. The recovery component has fewer tasks on the critical path, implying that there is higher parallelism during recovery than mitigation.

The histogram displayed in Figure 13 shows that there are 13 different template types in use in this runbook. Further, we find that `TrafficShift` is the most frequently used template. This is because most user traffic is delivered over HTTPS to our web servers, and hence manipulated by tuning weighs in our software load balancers.

Figure 14 plots the number of steps per tasks—observe that most tasks were executed in more than one step, and several were paced in more than 10 steps, during both the mitigation and recovery phases.

5.3 Efficiency

We leverage drain tests to estimate how fast we are able to mitigate a real disaster. In this section, we focus on the scenario where an entire datacenter fails. Table 5 shows the time taken by Maelstrom to drain and restore traffic

Phases (Traffic Shift)	Time Duration	
	Maelstrom	Sequential
Drain web traffic	10 min	×1
Drain all user-facing traffic	40 min	×6.6
Drain all service traffic	110 min	×6.3
Restore web traffic	50 min	×1
Restore internal service traffic	1.5 hour	×4.3
Restore all service traffic	2 hour	×5.6

Table 5: Time duration of draining and restoring traffic of a datacenter. The data are collected from a real disaster for which we drained all the traffic out of the entire datacenter (and restored it after repair).

in a datacenter in different phases. The traffic of Facebook's main web service, referred to as *web traffic* in Figure 11 and Table 5, is used as the baseline. It takes less than a minute to propagate a change of drain multiplier (cf. §5.2.1) to Edge LBs when draining web traffic. Maelstrom typically does not drain web traffic in one step but gradually adjusts speed based on health monitoring. It takes about 10 minutes to fully drain web traffic, and 50 minutes to restore it. Restoration is slow as we wish to minimize backend overload due to cold caches. Overall, it takes 40 minutes to drain all the user-facing traffic, and 110 minutes to drain all service traffic including the traffic of internal systems.

We next evaluate whether Maelstrom provides an efficient methodology to mitigate and recover from a datacenter failure. We calculate the time needed to drain and restore the datacenter sequentially by summing up the time used by the traffic drain and restoration of every service in a runbook. As shown in Table 5, sequential drain and restoration would take up to 6× longer than Maelstrom. The results verify the efficiency of Maelstrom and demonstrate the importance of parallelizing operations.

Note that restoring traffic back to a datacenter encounters a narrower bottleneck where a single target datacenter is receiving more load, in comparison to draining traffic from a datacenter to many others. We prioritize restoring user-facing traffic back into the datacenter as this minimizes the risk of exposing users to multiple independent datacenter failures.

6 Experience

Drain tests help us understand interactions amongst systems in our complex infrastructure. We find drain tests to be one of the most efficient ways to understand how a system fits into our infrastructure. A successful drain test is a validation of our tooling which tracks inter-system dependencies and health monitoring, while a failed drain test reveals gaps in our understanding. We find that drain tests are truer validators of inter-service dependencies than other methods we have experimented with, such as methods based on log and trace analysis.

Drain tests help us prepare for disaster. Prior to running regular drain tests, we often encountered delays in disaster mitigation due to our tools having atrophied as they did not account for evolving software, configuration and shared infrastructure components. Drain tests exercise our tooling continuously and confirm operational behavior in a controlled manner.

Drain tests are challenging to run. We observe that infrastructure changes, new dependencies, software regressions, bugs and various other dynamic variables inevitably trigger unexpected issues during a drain test. We strive to continually tune and improve our monitoring systems to quickly assess impact and remediate issues. Further, we have focused on communication and continually educate engineering teams at Facebook on our tests and their utility so our systems are prepared.

Automating disaster mitigation completely is not a goal. Our initial aspiration was to take humans out of the loop when mitigating disasters. However, we have learned that it is prohibitively difficult to encode the analytical and decision making skills of human operators without introducing tremendous complexity. The current design of Maelstrom is centered around helping operators triage a disaster and efficiently mitigate it using our tools and well-tested strategies. We intentionally expose runtime states of each task and allow human operators to override operations. This strategy has proved simpler and more reliable than attempting to automate everything. Our experience with operators confirms that Maelstrom significantly reduces operational overhead and the errors that are inevitable in a manual mitigation strategy.

Building the right abstractions to handle failures is important, but takes time and iteration. We have evolved Maelstrom's abstractions to match the mental model of the teams whose systems are managed by Maelstrom. We find that our separation of runbooks and tasks allows each team to focus on maintaining their own service-specific policies without the need to (re-)build mechanisms. This separation also allows us to efficiently onboard new services, and ensure a high quality bar for task implementation. Lastly, we find that as new systems are onboarded, we need to create new task templates and other supporting extensions to satisfy their needs.

7 Limitation and Discussion

Maelstrom, and draining traffic in general to respond to outages, is not a panacea. In fact, we find that there is no single approach or mechanism that can mitigate all the failures that might affect a large-scale Internet service.

Capacity planning is critical to ensure that healthy datacenters and shared infrastructure like backbone and the datacenter network fabric have sufficient headroom to serve traffic from a failing datacenter. Drain tests can

help validate capacity plans but shortfalls can still be difficult to address as it takes time to purchase, deliver, and turn-up machines and network capacity. If a capacity shortfall were to exist, it is wholly possible that draining traffic from a failing datacenter might overwhelm healthy datacenters and trigger cascading failures. Our strategy is to work in lockstep with capacity planning, and also regularly perform drills (storm tests) that isolate one or more datacenters and confirm that the remaining capacity can serve all our user and service needs.

If an outage is triggered by malformed client requests, or a malicious payload, redirecting traffic away from a failing datacenter to healthy ones will spread the failure. We handle this scenario by applying traffic shifts in multiple steps; the first step is intentionally small so we can monitor all systems in the target datacenter and confirm their health before initiating a large-scale drain.

Traffic drains may not always be the fastest mitigation strategy. Specifically, outages triggered by buggy software or configuration changes might be mitigated faster by reverting suspect changes. We expect operators to decide which mitigation strategy to use.

8 Related Work

Many prior papers study failures and outages in large scale systems running on cloud infrastructure [18,22–24, 27, 33, 35, 37, 38, 56]. These papers share several common conclusions: 1) outage is inevitable at scale when systems are exposed to a myriad set of failure scenarios, 2) large-scale, complex systems cannot be completely modeled for reliability analysis, and thus failure response cannot be predicted in advance; and 3) the philosophy of building and operating highly-available services is to anticipate disasters and proactively prepare for them. We agree with these conclusions and built Maelstrom to mitigate and recover from failures.

Many prior studies have focused on fast recovery [11, 14, 38, 40, 41] and efficient diagnosis [9, 15, 31, 57, 58]. While these studies help resolve the root cause of failures and outages in a timely manner, our experience shows that even this speedy resolution exposes users to a frustrating experience. We use Maelstrom to mitigate failure and reduce user-visible impact, which buys us time for thorough diagnosis and recovery.

Fault-injection testing has been widely adopted to continuously exercise the fault tolerance of large-scale systems [2–5,8,21,30,49]. Maelstrom is *not* a fault-injection tool like Chaos Monkey [8, 49]. Specifically, Maelstrom is not designed for simulating machine- or component-level failures, but rather for responding to disastrous failures at the datacenter level.

Drain tests are different from annual, multi-day testing drills such as DiRT [29] and GameDay [42]. Fundamentally, drain tests focus on testing mitigation and recovery for user traffic and services without fully isolating or shutting down a datacenter. Drain tests are fully automated and run frequently. In contrast, DiRT and GameDay intentionally disconnect or shutdown one or more datacenters fully and exercise the entire technical and operational spectrum, including detection, mitigation, escalation, and recovery components of a response strategy. Aside: we also use Maelstrom in our own periodic large-scale, DiRT-like drills to verify the capability and end-to-end effectiveness of our disaster response strategies.

Kraken and TrafficShifter [32, 50] leverage live traffic for load testing to identify resource utilization bottlenecks; TrafficShift [28] can also drain stateless web traffic. Maelstrom uses similar underlying traffic management primitives to Kraken (cf. §2), and goes beyond TrafficShift in its capability to drain different traffic types, track dependencies, and order operations.

Traffic draining has been anecdotally mentioned as a method for mitigating failures for site reliability [10, 28, 29]. To our knowledge, existing systems only work with one service or one type of traffic, and cannot drain different types of traffic of heterogenous services. Maelstrom serves as the sole system for draining and restoring *all* the services in *all* the datacenters at Facebook.

9 Conclusion

As our infrastructure grows, we have learned that it is critical to develop trusted tools and mechanisms to prepare for and respond to failure. We describe Maelstrom which we have built and improved over the past four years to handle datacenter-level disasters. Maelstrom tracks dependencies amongst services and uses feedback loops to handle outages safely and efficiently. We propose drain tests as a new testing strategy to identify dependencies amongst services, and ensure that tools and procedures for handling failures are always up to date.

Much of the focus of Maelstrom has been around ensuring that Facebook stays available when an incident affects an entire datacenter. In practice, we find that many incidents affect only a subset of hardware and software systems rather than entire datacenters. Our next focus is on building tools to isolate outages to the minimal subset of the systems they affect.

Acknowledgments

We thank the reviewers and our shepherd, Justine Sherry, for comments that improved this paper. We thank Chunqiang Tang for insightful feedback on an early draft. Much of our work on Disaster Readiness, and drain tests in particular, would not be possible without the support of engineering teams across Facebook. We thank the numerous engineers who have helped us understand various systems, given us feedback on the tooling, monitoring, and methodology of Maelstrom, and helped us improve the reliability of our infrastructure.

References

[1] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, USA, Nov. 2016).

[2] ALLSPAW, J. Fault Injection in Production: Making the case for resilience testing. *Communications of the ACM (CACM) 55*, 10 (Oct. 2012), 48–52.

[3] ALVARO, P., ANDRUS, K., SANDEN, C., ROSENTHAL, C., BASIRI, A., AND HOCHSTEIN, L. Automating Failure Testing Research at Internet Scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (Santa Clara, CA, USA, Oct. 2016).

[4] ALVARO, P., ROSEN, J., AND HELLERSTEIN, J. M. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)* (Melbourne, Victoria, Australia, May 2015).

[5] ALVARO, P., AND TYMON, S. Abstracting the Geniuses Away from Failure Testing. *Communications of the ACM (CACM) 61*, 1 (Jan. 2018), 54–61.

[6] ARAÚJO, J. T., SAINO, L., BUYTENHEK, L., AND LANDA, R. Balancing on the Edge: Transport Affinity without Network State. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Renton, WA, USA, Apr. 2018).

[7] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*, 2 ed. Morgan and Claypool Publishers, 2013.

[8] BASIRI, A., BEHNAM, N., DE ROOIJ, R., HOCHSTEIN, L., KOSEWSKI, L., REYNOLDS, J., AND ROSENTHAL, C. Chaos Engineering. *IEEE Software 33*, 3 (May 2016), 35–41.

[9] BESCHASTNIKH, I., WANG, P., BRUN, Y., AND ERNST, M. D. Debugging Distributed Systems. *Communications of the ACM (CACM) 59*, 8 (Aug. 2016), 32–37.

[10] BEYER, B., JONES, C., PETOFF, J., AND MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media Inc., 2016.

[11] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)* (San Antonio, TX, USA, June 2003).

[12] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Communications of the ACM (CACM) 59*, 5 (May 2016), 50–57.

[13] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)* (Cascais, Portugal, Oct. 2011).

[14] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, USA, Dec. 2004).

[15] CHEN, A., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)* (Belgrade, Serbia, Apr. 2017).

[16] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, USA, Oct. 2014).

[17] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)* (Santa Clara, CA, USA, Mar. 2016).

[18] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, Oct. 2010).

[19] GOOGLE CLOUD. Disaster Recovery Cookbook, 2017. https://cloud.google.com/solutions/disaster-recovery-cookbook.

[20] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)* (Florianópolis, Brazil, Aug. 2016).

[21] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Boston, MA, USA, Mar. 2011).

[22] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, USA, Nov. 2014).

[23] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (Santa Clara, CA, USA, Oct. 2016).

[24] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Oakland, CA, USA, Feb. 2018).

[25] GUPTA, A., AND SHUTE, J. High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads. In *Proceedings of the 9th Workshop on Business Intelligence for the Real Time Enterprise (BIRTE'15)* (Kohala Coast, HI, USA, Aug. 2015).

[26] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., VENKATARAMAN, V., VEERARAGHAVAN, K., AND SONG, Y. J. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Shanghai, China, Oct. 2017).

[27] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)* (San Francisco, CA, USA, Mar. 2002).

[28] KEHOE, M., AND MALLAPUR, A. TrafficShift: Avoiding Disasters at Scale. In *USENIX SRECon'17 Americas* (San Francisco, CA, USA, Mar. 2017).

[29] KRISHNAN, K. Weathering the Unexpected. *Communications of the ACM (CACM) 55*, 11 (Nov. 2012), 48–52.

[30] LEESATAPORNWONGSA, T., AND GUNAWI, H. S. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, USA, Nov. 2014).

[31] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th Conference on Symposium on Networked Systems Design and Implementation (NSDI'08)* (San Francisco, CA, USA, Apr. 2008).

[32] MALLAPUR, A., AND KEHOE, M. TrafficShift: Load Testing at Scale, May 2017. https://engineering.linkedin.com/blog/2017/05/trafficshift--load-testing-at-scale.

[33] MAURER, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM (CACM) 58*, 11 (Nov. 2015), 44–49.

[34] MEZA, J., XU, T., VEERARAGHAVAN, K., AND SONG, Y. J. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the 2018 ACM Internet Measurement Conference (IMC'18)* (Boston, MA, USA, Oct. 2018).

[35] MOGUL, J. C., ISAACS, R., AND WELCH, B. Thinking about Availability in Large Service Infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)* (Whistler, BC, Canada, May 2017).

[36] OLTEANU, V., AGACHE, A., VOINESCU, A., AND RAICIU, C. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Renton, WA, USA, Apr. 2018).

[37] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Seattle, WA, USA, Mar. 2003).

[38] PATTERSON, D., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, University of California Berkeley, Mar. 2002.

[39] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A Fast, Scalable, In-Memory Time Series Database. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB'15)* (Kohala Coast, HI, USA, Aug. 2015).

[40] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs As Allergies — A Safe Method to Survive Software Failure. In *Proceedings of the 20th Symposium on Operating System Principles (SOSP'05)* (Brighton, United Kingdom, Oct. 2005).

[41] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, USA, Dec. 2004).

[42] ROBBINS, J., KRISHNAN, K., ALLSPAW, J., AND LIMONCELLI, T. Resilience Engineering: Learning to Embrace Failure. *ACM Queue 10*, 9 (Sept. 2012), 1–9.

[43] SCHLINKER, B., KIM, H., CUI, T., KATZ-BASSETT, E., MADHYASTHA, H. V., CUNHA, I., QUINN, J., HASAN, S., LAPUKHOV, P., AND ZENG, H. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)* (Los Angeles, CA, USA, Aug. 2017).

[44] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)* (Prague, Czech Republic, Apr. 2013).

[45] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. ACMS: The Akamai Configuration Management System. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI'05)* (Boston, MA, USA, May 2005).

[46] SOMMERMANN, D., AND FRINDELL, A. Introducing Proxygen, Facebook's C++ HTTP framework, Nov. 2014. https://code.facebook.com/posts/1503205539947302.

[47] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Monterey, CA, USA, Oct. 2015).

[48] TREYNOR, B., DAHLIN, M., RAU, V., AND BEYER, B. The Calculus of Service Availability. *Communications of the ACM (CACM) 60*, 9 (Sept. 2017), 42–47.

[49] TSEITLIN, A. The Antifragile Organization. *Communications of the ACM (CACM) 56*, 8 (August 2013), 40–44.

[50] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, USA, Nov. 2016).

[51] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Bordeaux, France, Apr. 2015).

[52] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, USA, Nov. 2016).

[53] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP'13)* (Farmington, PA, USA, Nov. 2013).

[54] YAP, K.-K., MOTIWALA, M., RAHE, J., PADGETT, S., HOLLIMAN, M., BALDUS, G., HINES, M., KIM, T., NARAYANAN, A., JAIN, A., LIN, V., RICE, C., ROGAN, B., SINGH, A., TANAKA, B., VERMA, M., SOOD, P., TARIQ, M., TIERNEY, M., TRUMIC, D., VALANCIUS, V., YING, C., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)* (Los Angeles, CA, USA, Aug. 2017).

[55] YIN, Z., YUAN, D., ZHOU, Y., PASUPATHY, S., AND BAIRAVASUNDARAM, L. N. How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'11)* (Szeged, Hungary, Sept. 2011).

[56] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, USA, Oct. 2014).

[57] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., TANG, X., ZHOU, Y., AND SAVAGE, S. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, USA, Oct. 2012).

[58] ZHANG, Y., MAKAROV, S., REN, X., LION, D., AND YUAN, D. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Shanghai, China, Oct. 2017).

Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems

Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison

Abstract

We introduce *situation-aware updates and crash recovery* (SAUCR), a new approach to performing replicated data updates in a distributed system. SAUCR adapts the update protocol to the current situation: with many nodes up, SAUCR buffers updates in memory; when failures arise, SAUCR flushes updates to disk. This situation-awareness enables SAUCR to achieve high performance while offering strong durability and availability guarantees. We implement a prototype of SAUCR in ZooKeeper. Through rigorous crash testing, we demonstrate that SAUCR significantly improves durability and availability compared to systems that always write only to memory. We also show that SAUCR's reliability improvements come at little or no cost: SAUCR's overheads are within 0%-9% of a purely memory-based system.

1 Introduction

The correctness and performance of a fault-tolerant system depend, to a great extent, upon its underlying replication protocols. In the modern data center, these protocols include Paxos [45], Viewstamped Replication [48], Raft [57], and ZAB [39]. If these protocols behave incorrectly, reliability goals will not be met; if they perform poorly, excess resources and cost will be incurred.

A key point of differentiation among these protocols relates to how they store system state (§2). In one approach, which we call *disk durable*, critical state is replicated to persistent storage (i.e., hard drives or SSDs) within each node of the system [13, 14, 17, 37, 57]. In the contrasting *memory durable* approach, the state is replicated only to the (volatile) memory of each machine [48, 55]. Unfortunately, neither approach is ideal.

With the disk-durable approach, safety is paramount. When correctly implemented, by committing updates to disks within a majority of nodes, the disk-durable approach offers excellent durability and availability. Specifically, data will not be lost if the nodes crash and recover; further, the system will remain available if a bare majority of nodes are available. Unfortunately, the cost of safety is performance. When forcing updates to hard drives, disk-durable methods incur a 50× overhead; even when using flash-based SSDs, the cost is high (roughly 2.5×).

With the memory-durable approach, in contrast, performance is generally high, but at a cost: durability. In the presence of crash scenarios where a majority of nodes crash (and then recover), existing approaches can lead to data loss or indefinite unavailability.

The distributed system developer is thus confronted with a vexing quandary: choose safety and pay a high performance cost, or choose performance and face a potential durability problem. A significant number of systems [17, 41, 48, 55, 61] lean towards performance, employing memory-durable approaches and thus risking data loss or unavailability. Even when using a system built in the disk-durable manner, performance concerns can entice the unwary system administrator towards disaster; for instance, the normally reliable disk-durable ZooKeeper can be configured to run in a memory-durable mode [5], leading (regrettably) to data loss [30].

In this paper, we address this problem by introducing *situation-aware updates and crash recovery* or SAUCR (§3), a hybrid replication protocol that aims to provide the high performance of memory-durable techniques while offering strong guarantees similar to disk-durable approaches. The key idea underlying SAUCR is that the mode of replication should depend upon the situation the distributed system is in at a given time. In the common case, with many (or all) nodes up and running, SAUCR runs in memory-durable mode, thus achieving excellent throughput and low latency; when nodes crash or become partitioned, SAUCR transitions to disk-durable operation, thus ensuring safety at a lower performance level.

SAUCR applies several techniques to achieve high performance and safety. For example, a *mode-switch technique* enables SAUCR to transition between the fast memory-durable and the safe disk-durable modes. Next, given that SAUCR can operate in two modes, a node recovering from a crash performs *mode-aware crash recovery*; the node recovers the data from either its local disk or other nodes depending on its pre-crash mode. Finally, to enable a node to safely recover from a fast-mode crash, the other nodes store enough information about the node's state within them in the form of *replicated last-logged entry (LLE) maps*.

The effectiveness of SAUCR depends upon the simultaneity of failures. Specifically, if a window of time ex-

ists between individual node failures, the system can detect and thus react to failures as they occur. SAUCR takes advantage of this window in order to move from its fast mode to its slow-and-safe mode.

With *independent* failures, such a time gap between failures exists because the likelihood of many nodes failing together is negligible. Unfortunately, failures can often be *correlated* as well, and in that case, many nodes can fail together [31, 35, 42, 64]. Although many nodes fail together, a correlated failure does not necessarily mean that the nodes fail at the same instant: the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated failures, a time gap (ranging from a few milliseconds to a few seconds) exists between the individual failures; such a gap allows SAUCR to react to failures as they occur. With simultaneous failures, in contrast, such a window does not exist. However, we conjecture that such truly simultaneous failures are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). While we cannot definitively be assured of the veracity of NSC, our analysis (§2.3) of existing data [31, 33] hints at its likely truth.

Compared to memory-durable systems, SAUCR improves reliability under many failure scenarios. Under independent and non-simultaneous correlated failures, SAUCR always preserves durability and availability, offering the same guarantees as a disk-durable system; in contrast, memory-durable systems can lead to data loss or unavailability. Additionally, if NSC holds, SAUCR always provides the same guarantees as a disk-durable system. Finally, when NSC does not hold and if more than a majority of nodes crash in a truly simultaneous fashion, SAUCR remains unavailable, but preserves safety.

We implement (§4) and evaluate (§5) a prototype of SAUCR in ZooKeeper [4]. Through rigorous fault injection, we demonstrate that SAUCR remains durable and available in hundreds of crash scenarios, showing its robustness. This same test framework, when applied to existing memory-durable protocols, finds numerous cases that lead to data loss or unavailability. SAUCR's reliability improvements come at little or no performance cost: SAUCR's overheads are within 0%-9% of memory-durable ZooKeeper across six different YCSB workloads. Compared to the disk-durable ZooKeeper, with a slight reduction in availability in rare cases, SAUCR improves performance by 25× to 100× on HDDs and 2.5× on SSDs.

2 Distributed Updates and Recovery

In this section, we first describe the disk-durable and memory-durable protocols. We then describe the characteristics of different kinds of failures. Finally, we draw attention to the non-reactiveness to failures and the static nature of existing protocols.

	Mode	Avg. Latency (μs)	Throughput (ops/s)
HDD	`fsync`-s disabled	327.86	3050.1
cluster1	disk durability	16665.18 (50.8× ↑)	60.0 (50.8× ↓)
SSD	`fsync`-s disabled	461.2	2168.34
cluster2	disk durability	1027.3 (2.3× ↑)	973.4 (2.3× ↓)

Table 1: **Disk Durability Overheads.** *The table shows the overheads of disk durability. The experimental setup is detailed in §5.2.*

2.1 Disk-Durable Protocols

Disk-durable protocols always update the disk on a certain number of nodes upon every data modification. For example, ZooKeeper [4], etcd [24], and other systems [14, 49, 53, 62] persist every update on a majority of nodes before acknowledging clients.

With the exception of subtle bugs [2], disk-durable protocols offer excellent durability and availability. Specifically, committed data will never be lost under any crash failures. Further, as long as a majority of nodes are functional, the system will remain available. Unfortunately, such strong durability and availability guarantees come at a cost: poor performance.

Disk-durable protocols operate with caution and pessimistically flush updates to the disk (e.g., by invoking the `fsync` system call [11, 60]). Such forced writes in the critical path are expensive, often prohibitively so. To highlight these overheads, we conduct a simple experiment with ZooKeeper in the following modes: first, in the disk-durable configuration in which the `fsync` calls are enabled; second, with `fsync` calls disabled. A client sends update requests in a closed loop to the leader which then forwards the requests to the followers. We run the experiment on a five-node cluster and thus at least three servers must persist the data before acknowledgment.

As shown in Table 1, on HDDs, forced writes incur a 50× performance overhead compared to the `fsync`-disabled mode. Even on SSDs, the cost of forced writes is high (2.3×). While batching across many clients may alleviate some overheads, disk-durable protocols are fundamentally limited by the cost of forced writes and thus suffer from high latencies and low throughput.

A disk-durable update protocol is usually accompanied by a disk-based recovery protocol. During crash-recovery, a node can immediately join the cluster just after it recovers the data from its disk. A recovering node can completely trust its disk because the node would not have acknowledged any external entity before persisting the data. However, the node may be lagging: it may not contain some data that other nodes might have stored after it crashed. Even in such cases, the node can immediately join the cluster; if the node runs for an election, the leader-election protocol will preclude this node from becoming the leader because it has not stored some data that the other nodes have [2, 57]. If a leader already exists, the node fetches the missed updates from the leader.

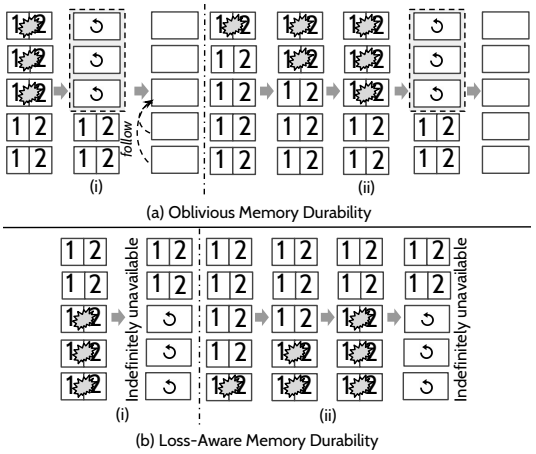


Figure 1: **Problems in Memory-Durable Approaches.** *(a) and (b) show how a data loss or an unavailability can occur with oblivious and loss-aware memory durability, respectively. In (i), the nodes fail simultaneously; in (ii), they fail non-simultaneously, one after the other.*

2.2 Memory-Durable Protocols

Given the high overheads imposed by a disk-durable protocol, researchers and practitioners alike have proposed alternative protocols [17, 55], in which the data is always buffered in memory, achieving good performance. We call such protocols *memory-durable* protocols.

2.2.1 Oblivious Memory Durability

The easiest way to achieve memory "durability" is *oblivious memory durability*, in which any forced writes in the protocol are simply disabled, unaware of the risks of only buffering the data in memory. Most systems provide such a configuration option [8, 22, 27, 62]; for example, in ZooKeeper, turning off the *forceSync* flag disables all `fsync` calls [5]. Turning off forced writes increases performance significantly, which has tempted practitioners to do so in many real-world deployments [29, 38, 59].

Unfortunately, disabling forced writes might lead to a data loss [5, 43] or sometimes even an unexpected data corruption [68]. Developers and practitioners have reported several instances where this unsafe practice has led to disastrous data-loss events in the real world [7, 30].

Consider the scenarios shown in Figure 1(a), in which ZooKeeper runs with *forceSync* disabled. If a majority of nodes crash and recover, data could be silently lost. Specifically, the nodes that crash could form a majority and elect a leader among themselves after recovery; however, this majority of nodes have lost their volatile state and thus do not know of the previously committed data, causing a silent data loss. The intact copies of data on other nodes (servers 4 and 5) can be overwritten by the new leader because the followers always follow the leader's state in ZooKeeper [2, 57].

2.2.2 Loss-Aware Memory Durability

Given that naïvely disabling forced writes may lead to a silent data loss, researchers have examined more careful approaches. In these approaches, a node, after a crash and a subsequent reboot, realizes that it might have lost its data; thus, a recovering node first runs a distinct recovery protocol. We call such approaches *loss-aware memory-durable* approaches.

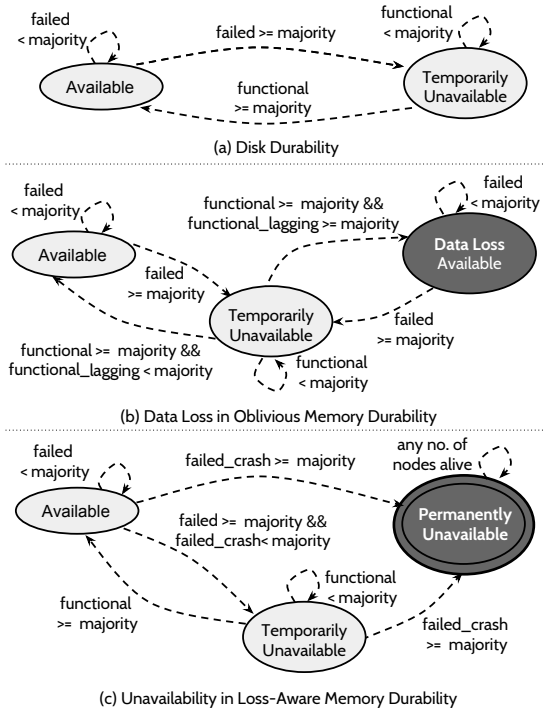


Figure 2: **Summary of Protocol Behaviors and Guarantees.** *The figure shows how the disk-durable and memory-durable protocols behave under failures and the guarantees they provide.*

The view-stamped replication (VR) protocol [55] is an example of this approach. Similarly, researchers at Google observed that they could optimize their Paxos-based system [17] by removing disk flushes, given that the nodes run a recovery protocol. For simplicity, we use only VR as an example for further discussion.

In VR, when a node recovers from a crash, it first marks itself to be in a *recovering* state, in which the node can neither participate in replication nor give votes to elect a new leader (i.e., a view change) [48]. Then, the node sends a recovery message to other servers. A node can respond to this message if it is *not* in the *recovering* state; the responding node sends its data to the recovering node. Once the node collects responses from a majority of servers (including the leader of the latest view), it can fix its data. By running a recovery protocol, this approach prevents a silent data loss.

Unfortunately, the loss-aware approach can lead to unavailability in many failure scenarios. Such an unavailability event could be *permanent*: the system may remain unavailable indefinitely even after all nodes have recovered from failures. For example, in Figure 1(b), a majority of nodes crash and recover. However, after re-

covering from the crash, none of the nodes will be able to collect recovery responses from a majority of nodes (because nodes in the *recovering* state cannot respond to the recovery messages), leading to permanent unavailability.

Protocols Summary. Figure 2 summarizes the behaviors of the disk-durable and memory-durable protocols. A node either could be functional or could have failed (crashed or partitioned). Disk-durable protocols remain available as long as a majority are functional. The system becomes *temporarily* unavailable if a majority fail; however, it becomes available once a majority recover. Further, the protocol is durable at all times.

The oblivious memory-durable protocol becomes temporarily unavailable if a majority fail. After recovering from a failure, a node could be lagging: it either recovers from a crash, losing all its data, or it recovers from a partition failure, and so it may not have seen updates. If such functional but lagging nodes form a majority, the system can silently lose data.

The loss-aware memory-durable approach becomes temporarily unavailable if the system is unable to form a majority due to partitions. However, the system becomes permanently unavailable if a majority or more nodes crash at any point; the system cannot recover from such a state, regardless of how many nodes recover.

2.3 Failures and Failure Asynchrony

Given that existing approaches compromise on either performance or reliability, our goal is to design a distributed update protocol that delivers high performance while providing strong guarantees. Such a design needs a careful understanding of how failures occur in datacenter distributed systems, which we discuss next.

Similar to most distributed systems, our goal is to tolerate only fail-recover failures [34, 36, 45, 57] and not Byzantine failures [16, 46]. In the fail-recover model, nodes may fail any time and recover later. For instance, a node may crash due to a power loss and recover when the power is restored. When a node recovers, it loses all its volatile state and is left only with its on-disk data. We assume that persistent storage will be accessible after recovering from the crash and that it will not be corrupted [32]. In addition to crashing, sometimes, a node could be partitioned and may later be able to communicate with the other nodes; however, during such partition failures, the node does not lose its volatile state.

Sometimes, node failures are *independent*. For example, in large deployments, single-node failure events are often independent: a crash of one node (e.g., due to a power failure) does not affect some other node. It is unlikely for many such independent failures to occur together, especially given the use of strategies such as failure-domain-aware placement [3, 44, 50].

With independent failures, the likelihood that a majority of nodes fail together is negligible. Under such a condition, designing a protocol that provides both high performance and strong guarantees is fairly straightforward: the protocol can simply buffer updates in memory always. Given that a majority will not be down at any point, the system will always remain available. Further, at least one node in the alive majority will contain all the committed data, preventing a data loss.

Unfortunately, in reality, such a failure-independence assumption is rarely justified. In many deployments, failures can be correlated [12, 20, 25, 35, 64, 66]. During such correlated crashes, several nodes fail together, often due to the same underlying cause such as rolling reboots [31], bad updates [54], bad inputs [26], or data-center-wide power outages [42].

Given that failures can be correlated, it is likely that the above naïve protocol may lose data or become unavailable. An ideal protocol must provide good performance and strong guarantees in the presence of correlated failures. However, designing such a protocol is challenging. At a high level, if the updates are buffered in memory (aiming to achieve good performance), a correlated failure may take down all the nodes together, causing the nodes to lose the data, affecting durability.

Although many or all nodes fail together, a correlated failure does not mean that the nodes fail at the same instant; the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated crashes, a time gap between the individual node failures exists. For instance, a popular correlated-crash scenario arises due to bad inputs: many nodes process a bad input and crash together [26]. However, such a bad input is not applied at exactly the same time on all the nodes (for instance, a leader applies an input before its followers), causing the individual failures to be non-simultaneous.

In contrast, with simultaneous correlated crashes, such a window between failures does not exist; all nodes may fail before any node can detect a failure and react to it. However, we conjecture that such truly simultaneous crashes are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). Publicly available data supports NSC. For example, a study of failures in Google data centers [31] showed that in most correlated failures, nodes fail one after the other, usually a few seconds apart.

We also analyze the time gap between failures in the publicly available Google cluster data set [33]. This data set contains traces of machine events (such as the times of node failures and restarts) of about 12K machines over 29 days and contains about 10K failure events. From the traces, we randomly pick five machines (without considering failure domains) and examine the timestamps of their failures. We repeat this 1M times (choosing different sets of machines). We find that the time between two failures among the picked machines is greater than 50

ms in 99.9999% of the cases. However, we believe the above percentage is a conservative estimate, given that we did not pick the machines across failure domains; doing so is likely to increase the time between machine failures. Thus, we observe that truly simultaneous machine failures are rare: a gap of 50 ms or more almost always exists between the individual failures.

Given that in most (if not all) failure scenarios, a window of time exists between the individual failures, a system can take advantage of the window to react and perform a preventive measure (e.g., flushing to disk). A system that exploits this asynchrony in failures can improve durability and availability significantly.

2.4 Non-Reactiveness and Static Nature

We observe that existing update protocols do not react to failures. While it may be difficult to react to truly simultaneous failures, with independent and non-simultaneous failures, an opportunity exists to detect failures and perform a corrective step. However, existing protocols do not react to *any* failure.

For example, the oblivious memory-durable protocol can lose data, regardless of the simultaneity of the failures. Specifically, a data loss occurs both in Figure 1(a)(i) in which the nodes crash simultaneously and (a)(ii) in which they fail non-simultaneously. Similarly, the loss-aware approach can become unavailable, regardless of the simultaneity of the failures (as shown in Figure 1(b)). This is the reason we do not differentiate simultaneous and non-simultaneous failures in Figure 2; the protocols behave the same under both failures.

Next, we note that the protocols are *static* in nature: they always update and recover in a constant way, regardless of the situation; this situation-obliviousness is the cause for poor performance or reliability. For example, the disk-durable protocol constantly anticipates failures, forcing writes to disk even when nodes never or rarely crash; this unnecessary pessimism leads to poor performance. In contrast, when nodes rarely crash, a situation-aware approach would buffer updates in memory, achieving high performance. Similarly, the memory-durable protocol always optimistically buffers updates in memory even when only a bare majority are currently functional; this unwarranted optimism results in poor durability or availability. In contrast, when only a bare majority are alive, a situation-aware approach would safely flush updates to disk, improving durability and availability.

Our approach, *situation-aware updates and crash recovery* or SAUCR, reacts to failures quickly with corrective measures, and adapts to the current situation of the system. Such reactiveness and situation-awareness enables SAUCR to achieve high performance similar to a memory-durable protocol while providing strong guarantees similar to a disk-durable protocol.

3 Situation-Aware Updates and Recovery

The main idea in SAUCR is that of situation-aware operation, in which the system operates in two modes: *fast* and *slow*. In the common case, with many or all nodes up, SAUCR operates in the fast mode, buffering updates in memory and thus achieving high performance. When failures arise, SAUCR quickly detects them and performs two corrective measures. First, the nodes flush their data to disk, preventing an imminent data loss or unavailability. Second, SAUCR commits subsequent updates in slow mode, in which the nodes synchronously write to disk, sacrificing performance to improve reliability.

When a node recovers from a crash, it performs mode-aware recovery. The node recovers its data either from its local disk or from other nodes depending on whether it operated in slow or fast mode before it crashed.

We first outline SAUCR's guarantees (§3.1) and provide an overview of SAUCR's modes (§3.2). Next, we discuss how SAUCR detects and reacts to failures (§3.3), and describe the mechanisms that enable mode-aware recovery (§3.4). We then explain how crash recovery works and show its safety (§3.5). Finally, we summarize SAUCR's key aspects and describe the guarantees in detail (§3.6).

3.1 Guarantees

We consider three kinds of failures: independent, correlated non-simultaneous, and correlated simultaneous failures. SAUCR can tolerate any number of independent and non-simultaneous crashes; under such failures, SAUCR always guarantees durability. As long as a majority of servers eventually recover, SAUCR guarantees availability. Under simultaneous correlated failures, if a majority or fewer nodes crash, and if eventually a majority recover, SAUCR will provide durability and availability. However, if *more than* a majority crash simultaneously, then SAUCR cannot guarantee durability and so will remain unavailable. However, we believe such truly simultaneous crashes are extremely rare. We discuss the guarantees in more detail later (§3.6).

3.2 SAUCR Modes Overview

We first describe some properties common to many majority-based systems. We then highlight how SAUCR differs from existing systems in key aspects.

Most majority-based systems are *leader-based* [6, 57]; the clients send updates to the leader which then forwards them to the followers. The updates are first stored in a log and are later applied to an application-specific data structure. A leader is associated with an *epoch*: a slice of time; for any given epoch, there could be at most one leader [6, 57]. Because only the leader proposes an update, each update is uniquely qualified by the epoch in which the leader proposed it and the index of the update in the log. The leader periodically checks if a follower

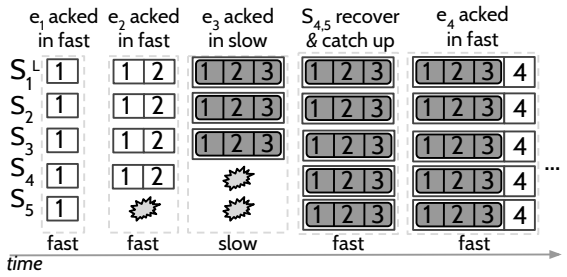


Figure 3: **Saucr Modes.** *The figure shows how* SAUCR*'s modes work. S_1 is the leader. Entries in a white box are committed but are only buffered (e.g., e_1 and e_2 in the first and second states). Entries shown grey denote that they are persisted (e.g., $e_1 - e_3$ in the third state). In fast mode, a node loses its data upon a crash and is annotated with a crash symbol (e.g., S_5 has lost its data in the second state).*

is alive or not via heartbeats. If the followers suspect that the leader has failed, they compete to become the new leader in a new epoch. Most systems guarantee the *leader-completeness* property: a candidate can become the leader only if it has stored all items that have been acknowledged as committed [2, 57]. SAUCR retains all the above properties of majority-based systems.

In a memory-durable system, the nodes always buffer updates in memory; similarly, the updates are always synchronously persisted in a disk-durable system. SAUCR changes this fundamental attribute: SAUCR either buffers the updates or synchronously flushes them to disk depending on the situation. When more nodes than a bare minimum to complete an update are functional, losing those additional nodes will not result in an immediate data loss or unavailability; in such situations, SAUCR operates in fast mode. Specifically, SAUCR operates in fast mode if *more than* a bare majority are functional (i.e., functional $\geq \lceil n/2 \rceil + 1$, where n is the total nodes, typically a small odd number). If nodes fail and only a bare majority ($\lceil n/2 \rceil$) are functional, losing even one additional node may lead to a data loss or unavailability; in such situations, SAUCR switches to the slow mode. Because the leader continually learns about the status of the followers, the leader determines the mode in which a particular request must be committed.

We use Figure 3 to give an intuition about how SAUCR's modes work. At first, all the nodes are functional and hence the leader S_1 replicates entry e_1 in fast mode. The followers acknowledge e_1 before persisting it (before invoking `fsync`); similarly, the leader also only buffers e_1 in memory. In fast mode, the leader acknowledges an update only after $\lceil n/2 \rceil + 1$ nodes have buffered the update. Because at least four nodes have buffered e_1, the leader acknowledges e_1 as committed. Now, S_5 crashes; the leader detects this but remains in fast mode and commits e_2 in fast mode.

Next, S_4 also crashes, leaving behind a bare majority; the leader now immediately initiates a switch to slow mode and replicates all subsequent entries in slow mode. Thus, e_3 is replicated in slow mode. Committing an entry in slow mode requires at least a bare majority to persist the entry to their disks; hence, when e_3 is persisted on three nodes, it is committed. Further, the first entry persisted in slow mode also persists all previous entries buffered in memory; thus, when e_3 commits, e_1 and e_2 are also persisted. Meanwhile, S_4 and S_5 recover and catch up with other nodes; therefore, the leader switches back to fast mode, commits e_4 in fast mode, and continues to commit entries in fast mode until further failures.

3.3 Failure Reaction

In the common case, with all or many nodes alive, SAUCR operates in fast mode. When failures arise, the system needs to detect them and switch to slow mode or flush to disk. The basic mechanism SAUCR uses to detect failures is that of heartbeats.

Follower Failures and Mode Switches. If a follower fails, the leader detects it via missing heartbeats. If the leader suspects that only a bare majority (including self) are functional, it immediately initiates a switch to slow mode. The leader sends a special request (or uses an outstanding request such as e_3 in the above example) on which it sets a flag to indicate to the followers that they must respond only after persisting the request; this also ensures that all previously buffered data will be persisted. All subsequent requests are then replicated in slow mode. When in fast mode, the nodes periodically flush their buffers to disk in the background, without impacting the client-perceived performance. These background flushes reduce the amount of data that needs to be written when switching to slow mode. Once enough followers recover, the leader switches back to fast mode. To avoid fluctuations, the leader switches to fast mode after confirming a handful number of times that it promptly gets a response from more than a bare majority; however, a transition to slow mode is immediate: the first time the leader suspects that only a bare majority of nodes are alive.

Leader Failures and Flushes. The leader takes care of switching between modes. However, the leader itself may fail at any time. The followers quickly detect a failed leader (via heartbeats) and flush all their buffered data to disk. Again, the periodic background flushes reduce the amount of data that needs to be written.

3.4 Enabling Safe Mode-Aware Recovery

When a node recovers from a crash, it may have lost some data if it had operated in fast mode; in this case, the node needs to recover its lost data from other nodes. In contrast, the node would have all the data it had logged on its disk if it had crashed in slow mode or if it had flushed after detecting a failure; in such cases, it recovers the data only from its disk. Therefore, a recovering node first needs to determine the mode in which it last

operated. Moreover, if a node recovers from a fast-mode crash, the other nodes should maintain enough information about the recovering node. We now explain how SAUCR satisfies these two requirements.

3.4.1 Persistent Mode Markers

The SAUCR nodes determine their pre-crash mode as follows. When a node processes the first entry in fast mode, it synchronously persists the epoch-index pair of that entry to a structure called the *fast-switch-entry*. Note that this happens only for the first entry in the fast mode. In the slow mode or when flushing on failures, in addition to persisting the entries, the nodes also persist the epoch-index pair of the latest entry to a structure called the *latest-on-disk-entry*. To determine its pre-crash mode, a recovering node compares the above two on-disk structures. If its *fast-switch-entry* is ahead[1] of its *latest-on-disk-entry*, then the node concludes that it was in the fast mode. Conversely, if the *fast-switch-entry* is behind the *latest-on-disk-entry*, then the node concludes that it was in the slow mode or it had safely flushed to disk.

3.4.2 Replicated LLE Maps

Once a node recovers from a crash, it must know how many entries it had logged in memory or disk before it crashed. We refer to this value as the *last logged entry* or LLE of that node. The LLE-recovery step is crucial because only if a node knows its LLE, it can participate in elections. Specifically, a candidate requests votes from other nodes by sending its LLE. A participant grants its vote to a candidate if the participant's LLE and current epoch are not ahead of the candidate's LLE and current epoch, respectively [57] (provided the participant had not already voted for another candidate in this epoch).

In a majority-based system, as long as a majority of nodes are alive, the system must be able to elect a leader and make progress [10, 57]. It is possible that the system only has a bare majority of nodes including the currently recovering node. Hence, it is crucial for a recovering node to immediately recover its LLE; if it does not, it cannot participate in an election or give its vote to other candidates, rendering the system unavailable.

If a node recovers from a slow-mode crash, it can recover its LLE from its disk. However, if a node recovers from a fast-mode crash, it would *not* have its LLE on its disk; in this case, it has to recover its LLE from other nodes. To enable such a recovery, as part of the replication request, the leader sends a map of the last (potentially) logged entry of each node to every node. The leader constructs the map as follows: when replicating an entry at index i in epoch e, the leader sets the LLE of all the functional followers and self to e.i and retains the last successful value of LLE for the crashed or partitioned followers. For instance, if the leader (say, S_1) is replicating an entry at index 10 in epoch e to S_2, S_3, and S_4, and if S_5 has crashed after request 5, then the map will be $\langle S_1$:e.10, S_2:e.10, S_3:e.10, S_4:e.10, S_5:e.5$\rangle$. We call this map the *last-logged entry map* or LLE-MAP. In the fast mode, the nodes maintain the LLE-MAP in memory; in slow mode, the nodes persist the LLE-MAP to the disk.

[1] An entry a is ahead of another entry b if (a.epoch $>$ b.epoch) or (a.epoch $==$ b.epoch and a.index $>$ b.index).

3.5 Crash Recovery

In a disk-durable system, a node recovering from a crash performs three distinct recovery steps. First, it recovers its LLE from its disk. Second, it competes in an election with the recovered LLE. The node may either become the leader or a follower depending on its LLE's value. Third, the node recovers any missed updates from other nodes. If the node becomes the leader after the second step, it is guaranteed to have all the committed data because of the leader-completeness property [2, 57], skipping the third step. If the node becomes a follower, it might be lagging and so fetches the missed updates from the leader.

In SAUCR, a node recovering from a crash could have operated either in slow or fast mode before it crashed. If the node was in slow mode, then its recovery steps are identical to the disk-durable recovery described above; we thus do not discuss slow-mode crash recovery any further. A fast-mode crash recovery, however, is more involved. First, the recovering node would not have its LLE on its disk; it has to carefully recover its LLE from the replicated LLE-MAPs on other nodes. Second, it has to recover its lost data irrespective of whether it becomes the leader or a follower. We explain how a node performs the above crash-recovery steps.

Max-Among-Minority. A SAUCR node recovering from a fast-mode crash recovers its LLE using a procedure that we call *max-among-minority*. In this procedure, the node first marks itself to be in a state called *recovering* and then sends an LLE query to all other nodes. A node may respond to this query only if it is in a recovered (*not recovering*) state; if it is not, it simply ignores the query. Note that a node can be in the recovered state in two ways. First, it could have operated in fast mode and not crashed yet; second, it could have last operated in slow mode and so has the LLE-MAP on its disk. The recovering node waits to get responses from at least a bare minority of nodes, where $bare\text{-}minority = \lceil n/2 \rceil - 1$; once the node receives a bare-minority responses, it picks the maximum among the responses as its LLE. Finally, the node recovers the actual data up to the recovered LLE. For now, we assume that at least a bare minority will be in recovered state; we soon discuss cases where only fewer than a bare minority are in the recovered state (§3.6).

We argue that the max-among-minority procedure guarantees safety, i.e., it does not cause a data loss. To do so, let us consider a node N that is recovering from a

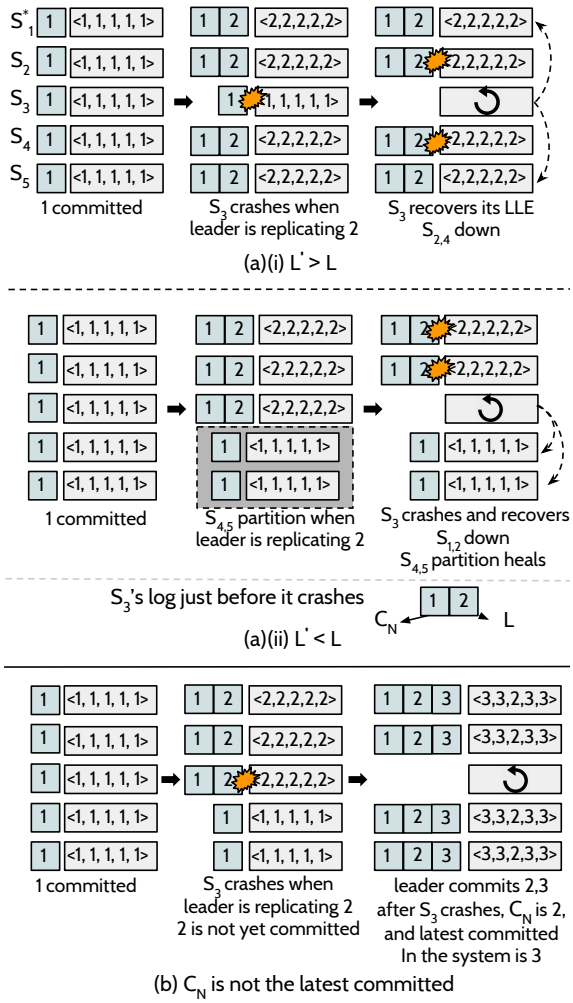


Figure 4: **LLE Recovery.** *The figure shows how L′ may not be equal to L. For each node, we show its log and LLE-MAP. The leader (S_1) is marked *; crashed nodes are annotated with a crash symbol; nodes partitioned are shown in a dotted box; epochs are not shown.*

fast-mode crash and let its actual last-logged entry (LLE) be L. When N runs the max-among-minority procedure, it retrieves L′ as its LLE and recovers all entries up to L′.

If N recovers exactly all entries that it logged before crashing (i.e., L′=L), then it is as though N had all the entries on its local disk (similar to how a node would recover in a disk-durable protocol, which is safe). Therefore, if the retrieved L′ is equal to the actual last-logged entry L, the system would be safe.

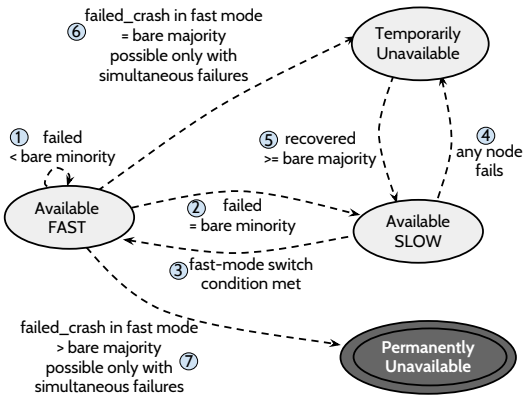
However, in reality, it may not be possible for N to retrieve an L′ that is exactly L. If N crashes after the leader sends a replication request but before N receives it, N may retrieve an L′ that is greater than L. For example, consider the case shown in Figure 4(a)(i). The leader (S_1) has successfully committed entry-1 in fast mode and now intends to replicate entry-2; hence, the leader populates the LLE-MAP with 2 as the value for all the nodes. However, S_3 crashes before it receives entry-2; consequently, its LLE is 1 when it crashed. However, when S_3 recovers its LLE from LLE-MAPs of S_1 and S_5 using the max-among-minority algorithm, the recovered L′ will be 2 which is greater than 1. Note that if L′ is greater than L, it means that N will recover additional entries that were not present in its log, which is safe. Similarly, it is possible for N to retrieve an L′ that is smaller than L. For instance, in Figure 4(a)(ii), S_3 has actually logged two entries; however, when it recovers, its L′ will be 1 which is smaller than the actual LLE 2. $L' < L$ is the only case that needs careful handling.

We now show that the system is safe even when the recovered L′ is smaller than L. We first establish a lower bound for L′ that guarantees safety. Then, we show that max-among-minority ensures that the recovered L′ is at least as high as the established lower bound.

Lower bound for L′. Let N's log when it crashed be D and let C_N be the last entry in D that is committed. For example, in Figure 4(a)(ii), for S_3, D contains entries 1 and 2, and the last entry in D that was committed is 1. Note that C_N need not be the latest committed entry; the system might have committed more entries after N crashed but none of these entries will be present in N's log. For example, in Figure 4(b), for S_3, C_N is 2 while the latest committed entry in the system is 3.

For the system to be safe, all *committed* entries must be recovered, while the *uncommitted* entries need *not* be recovered. For example, in Figure 4(a)(ii), it is safe if S_3 does not recover entry-2 because entry-2 is uncommitted. However, it is unsafe if N does not recover entry-1 because entry-1 is committed. For instance, imagine that S_3 runs an incorrect recovery algorithm that does not recover entry-1 in Figure 4(a)(ii). Now, if S_1 and S_2 also run the incorrect algorithm, then it is possible for S_1, S_2, and S_3 to form a majority and lose committed entry-1. Therefore, if the recovery ensures that N recovers all the entries up to C_N, committed data will not be lost, i.e., L′ must be at least as high as the last entry in N's log that is committed. In short, the lower bound for L′ is C_N. Next, we show that indeed the L′ recovered by max-among-minority is equal to or greater than C_N.

Proof Sketch for $L' \geq C_N$. We prove by contradiction. Consider a node N that is recovering from a fast-mode crash and that C_N is the last entry in N's log that was committed. During recovery, N queries a bare minority. Let us suppose that N recovers an L′ that is less than C_N. This condition can arise if a bare minority of nodes hold an LLE of N in their LLE-MAPs that is less than C_N. This is possible if the bare minority crashed long ago and recently recovered, or they were partitioned. However, if a bare minority had crashed or partitioned, it is *not* possible for the remaining bare majority to have committed C_N in fast mode (recall that a fast-mode commitment requires at least $\textit{bare-majority} + 1$ nodes to have bufferred



$bare\ majority = \lceil n/2 \rceil, bare\ minority = \lceil n/2 \rceil - 1$

Figure 5: **SAUCR Summary and Guarantees.** *The figure summarizes how* SAUCR *works under failures and the guarantees it provides.*

C_N and updated their LLE-MAPs). Therefore, C_N could have either been committed only in slow mode or not committed at all. However, if C_N was committed in slow mode, then N would be recovering from a slow-mode crash which contradicts the fact that N is recovering from a fast-mode crash. The other possibility that C_N could not have been committed at all directly contradicts the fact that C_N is committed. Therefore, our supposition that L' is less than C_N must be false.

Once a node has recovered its LLE, it can participate in elections. If an already recovered node or a node that has not failed so far becomes the leader (for example, S_1 or S_5 in Figure 4(a)(i)), it will already have the LLE-MAP, which it can use in subsequent replication requests. On the other hand, if a recently recovered node becomes the leader (for example, S_3 in Figure 4(a)(i)), then it needs to construct the LLE-MAP values for other nodes. To enable this construction, during an election, the voting nodes send their LLE-MAP to the candidate as part of the vote responses. Using these responses, the candidate constructs the LLE-MAP value for each node by picking the maximum LLE of that node from a bare-minority responses.

Data recovery. Once a node has successfully recovered its LLE, it needs to recover the actual data. If the recovering node becomes the follower, it simply fetches the latest data from the leader. In contrast, if the recovering node becomes the leader, it recovers the data up to the recovered LLE from the followers.

3.6 Summary and Guarantees

We use Figure 5 to summarize how SAUCR works and the guarantees it offers; a node fails either by crashing or by becoming unreachable over the network. We guide the reader through the description by following the sequence numbers shown in the figure. ① At first, we assume all nodes are in recovered state; in this state, SAUCR operates in the fast mode; when nodes fail, SAUCR stays in the fast mode as long as the number of nodes failed is less than a bare minority. ② After a bare minority of nodes fail, SAUCR switches to slow mode. ③ Once in slow mode, if one or more nodes recover and respond promptly for a few requests, SAUCR transitions back to fast mode. ④ In slow mode, if any node fails, SAUCR becomes temporarily unavailable. ⑤ Once a majority of nodes recover, the system becomes available again.

To explain further transitions, we differentiate non-simultaneous and simultaneous crashes and network partitions. In the presence of non-simultaneous crashes, nodes will have enough time to detect failures; the leader can detect follower crashes and switch to slow mode and followers can detect the leader's crash and flush to disk. Thus, despite any number of non-simultaneous crashes, SAUCR always transitions through slow mode. Once in slow mode, the system provides strong guarantees.

However, in the presence of simultaneous crashes, many nodes could crash instantaneously while in fast mode; in such a scenario, SAUCR cannot always transition through slow mode. ⑥ If the number of nodes that crash in fast mode does not exceed a majority, SAUCR will only be temporarily unavailable; in this case, at least a bare minority will be in recovered state or will have previously crashed in slow mode making crash recovery possible (as described in §3.5). ⑦ In rare cases, more than a bare majority of nodes may crash in fast mode, in which case, crash recovery is not possible: the number of nodes that are in recovered state or previously crashed in slow mode will be less than a bare minority. During such simultaneous crashes, which we believe are extremely rare, SAUCR remains unavailable.

In the presence of partitions, all nodes could be alive, but partitioned into two; in such a case, the minority partition would be temporarily unavailable while the other partition will safely move to slow mode if a bare majority are connected within the partition. The nodes in the minority partition would realize they are not connected to the leader and flush to disk. Both of these actions guarantee durability and prevent future unavailability.

4 Implementation

We have implemented situation-aware updates and crash recovery in Apache ZooKeeper (v3.4.8). We now describe the most important implementation details.

Storage layer. ZooKeeper maintains an on-disk log to which the updates are appended. ZooKeeper also maintains snapshots and meta information (e.g., current epoch). We modified the log-update protocol to not issue `fsync` calls synchronously in fast mode. Snapshots are periodically written to disk; because the snapshots are taken in the background, foreground performance is unaffected. The meta information is always synchronously updated. Fortunately, such synchronous updates happen rarely (only when the leader changes), and thus do

not affect common-case performance. In addition to the above structures, SAUCR maintains the *fast-switch-entry* in a separate file and synchronously updates it the first time when the node processes an entry in the fast mode. In slow mode, the LLE-MAP is synchronously persisted. SAUCR maintains the map at the head of the log file. The *latest-on-disk-entry* for a node is its own entry in the persistent LLE-MAP (LLE-MAP is keyed by node-id).

Replication. We modified the *QuorumPacket* [9] (which is used by the leader for replication) to include the mode flag and the LLE-MAP. The leader transitions to fast mode after receiving three consecutive successful replication acknowledgements from more than a bare majority.

Failure Reaction. In our implementation, the nodes detect failures through missing heartbeats, missing responses, and broken socket connections. Although quickly reacting to failures and flushing or switching modes is necessary to prevent data loss or unavailability, hastily declaring a node as failed might lead to instability. For example, if a follower runs for an election after missing just one heartbeat from the leader, the system may often change leader, affecting progress. SAUCR's implementation avoids this scenario as follows. On missing the first heartbeat from the leader, the followers *suspect* that the leader might have failed and so quickly react to the suspected failure by flushing their buffers. However, they conservatively wait for a handful of missing heartbeats before *declaring* the leader as failed and running for an election. Similarly, while the leader initiates a mode switch on missing the first heartbeat response, it waits for a few missing responses before declaring the follower as failed. If a majority of followers have not responded to a few heartbeats, the leader steps down and becomes a candidate.

Recovery Protocol. We modified the leader election protocol so that a node recovering from a fast-mode crash first recovers its LLE before it can participate in elections. A responding node correctly handles LLE-query from a node and replication requests from the leader that arrive concurrently. If a node that recovers from a fast-mode crash becomes the leader, it fetches the data items up to its LLE from others. However, due to the background flushes, several items might already be present on the disk; the node recovers only the missing items. The responding node batches several items in its response.

5 Evaluation

We now evaluate the durability, availability, and performance of our SAUCR implementation.

5.1 Durability and Availability

To evaluate the guarantees of SAUCR, we developed a cluster crash-testing framework. The framework first generates a graph of all possible cluster states as shown in Figure 6. Then, it generates a set of *cluster-state sequences*. For instance, $12345 \rightarrow 345 \rightarrow 45 \rightarrow 1245 \rightarrow 1 \rightarrow 13 \rightarrow 12345$ is one such sequence. In this sequence, at first, all five nodes are alive; then, two nodes (1 and 2) crash; then, 3 crashes; next, 1 and 2 recover; then 2, 4, 5 crash; 3 recovers; finally, 2, 4, 5 recover. To generate a sequence, we start from the root state where all nodes are alive. We visit a child with a probability that decreases with the length of the path constructed so far, and the difference in the number of alive nodes between the parent and the child. We produced 1264 such sequences (498 and 766 for a 5-node and 7-node cluster, respectively).

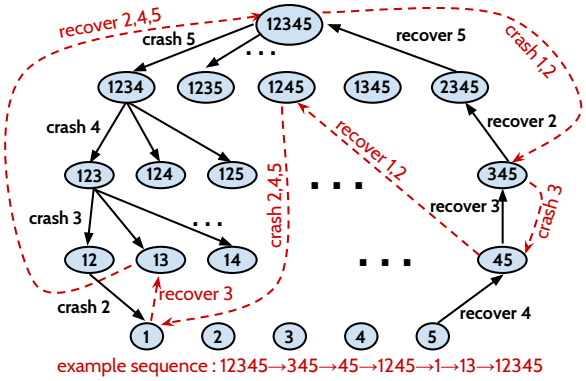


Figure 6: **Cluster-State Sequences.** *The figure shows the possible cluster states for a five-node cluster and how cluster-state sequences are generated. One example cluster-state sequence is traced.*

The cluster-state sequences help test multiple update and recovery code paths in SAUCR. For example, in the above sequence, 12345 would first operate in fast mode; then 345 would operate in slow mode; then 1245 would operate in fast mode; 1 would flush to disk on detecting that other nodes have crashed; in the penultimate state, 3 would recover from a slow-mode crash; in the last state, 2, 4, and 5 would recover from a fast-mode crash.

Within each sequence, at each intermediate cluster state, we insert new items if possible (if a majority of nodes do not exist, we cannot insert items). $12345^a \rightarrow 345^b \rightarrow 45 \rightarrow 1245^c \rightarrow 1 \rightarrow 13 \rightarrow 12345^d$ shows how entries *a-d* are inserted at various stages. In the end, the framework reads all the acknowledged items. If the cluster does not become available and respond to the queries, we flag the sequence as unavailable for the system under test. If the system silently loses the committed items, then we flag the sequence as data loss.

We subject the following four systems to the cluster-crash sequences: memory-durable ZK (ZooKeeper with the *forceSync* flag turned off), VR (viewstamped replication), disk-durable ZK (ZooKeeper with *forceSync* turned on), and finally SAUCR. Existing VR implementations [65] do not support a read/write interface, preventing us from directly applying our crash-testing framework to them. Therefore, we developed an *ideal* model of VR that resembles a perfect implementation.

System	Nodes	Non-simultaneous				Simultaneous				
		Total	Correct	Unavailable	Data loss	Scenario	Total	Correct	Unavailable	Data loss
ZK-mem	5	498	248	0	250	n/a	498	248	0	250
	7	766	455	0	311	n/a	766	455	0	311
VR-ideal	5	498	28	470	0	n/a	498	28	470	0
	7	766	189	577	0	n/a	766	189	577	0
ZK-disk	5	498	498	0	0	n/a	498	498	0	0
	7	766	766	0	0	n/a	766	766	0	0
SAUCR	5	498	498	0	0	other	475	475	0	0
						!min-rec	23	0	23	0
	7	766	766	0	0	other	725	725	0	0
						!min-rec	41	0	41	0

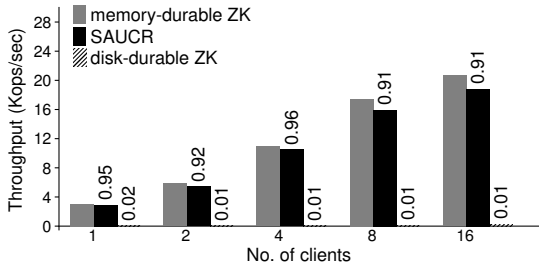
Table 2: **Durability and Availability.** *The table shows the durability and availability of memory-durable ZK (ZK-mem), VR (VR-ideal), disk-durable ZK (ZK-disk), and* SAUCR. *!min-rec denotes that only less than a bare minority are in recovered state.*

5.1.1 Non-simultaneous Crashes

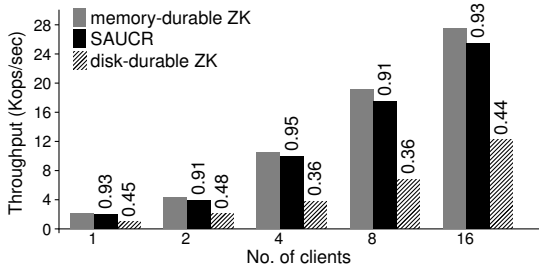
We first test all sequences considering that failures are non-simultaneous. For example, when the cluster transitions from 12345 to 345, we crash nodes 1 and 2 one after the other (with a gap of 50 ms). Table 2 shows the results. As shown, the memory-durable ZK loses data in about 50% and 40% of the cases in the 5-node and 7-node tests, respectively. The ideal VR model does not lose data; however, it leads to unavailability in about 90% and 75% of the cases in the 5-node and 7-node tests, respectively. As expected, disk-durable ZooKeeper is safe. In contrast to memory-durable ZK and VR, SAUCR remains durable and available in all cases. Because failures are non-simultaneous in this test, the leader detects failures and switches to slow mode; similarly, the followers quickly flush to disk if the leader crashes, leading to correct behavior.

5.1.2 Simultaneous Crashes

We next assume that failures are simultaneous. For example, if the cluster state transitions from 124567 to 12, we crash all four nodes at the same time, without any gap. Note that during such a failure, SAUCR would be operating in fast mode and suddenly many nodes would crash simultaneously, leaving behind less than a bare minority. In such cases, less than a bare minority would be in the *recovered* state; SAUCR cannot handle such cases. Table 2 shows the results. As shown, memory-durable ZK loses data in all cases in which it lost data in the non-simultaneous test. This is because memory-durable ZK loses data, irrespective of the simultaneity of the crashes. Similarly, VR is unavailable in all the cases where it was unavailable in the non-simultaneous crash tests. As expected, disk-durable ZK remains durable and available. SAUCR remains unavailable in a few cases by its design.



(a) HDD (cluster-1)



(b) SSD (cluster-2)

Figure 7: **Micro-benchmarks.** *(a) and (b) show the update throughput on memory-durable ZK,* SAUCR, *and disk-durable ZK on HDDs and SSDs, respectively. Each request is 1KB in size. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

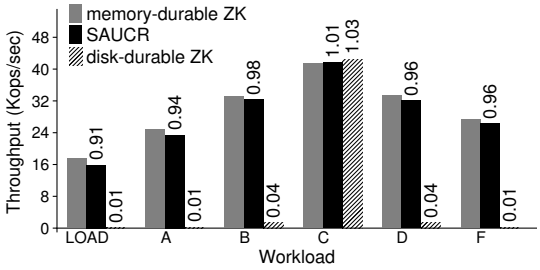
5.2 Performance

We conducted our performance experiments on two clusters (cluster-1: HDD, cluster-2: SSD), each with five machines. The HDD cluster has a 10 Gb network, and each node is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4, with a 1-TB HDD. The SSD cluster has 10 Gb network, and each node is a 20-core Intel E5-2660 machine with 160 GB memory running Linux 4.4, with a 480-GB SSD. Numbers reported are the average over five runs.

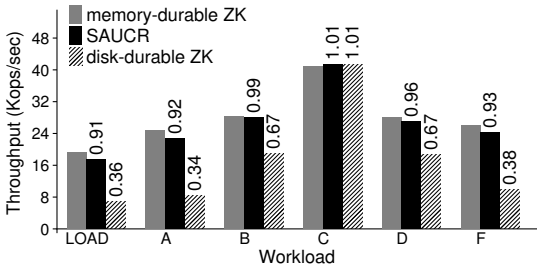
5.2.1 Update Micro-benchmark

We now compare SAUCR's performance against memory-durable ZK and disk-durable ZK. We conduct this experiment for an update-only micro-benchmark.

Figure 7(a) and (b) show the results on HDDs and SSDs, respectively. As shown in the figure, SAUCR's performance is close to the performance of memory-durable ZK (overheads are within 9% in the worst case). Note that SAUCR's performance is close to memory-durable ZK but not equal; this small gap exists because, in the fast mode, SAUCR commits a request only after four nodes (majority + 1) acknowledge, while memory-durable ZK commits a request after three nodes (a bare majority) acknowledge. Although the requests are sent to the followers in parallel, waiting for acknowledgment from one additional follower adds some delay. Compared to disk-durable ZK, as expected, both memory-durable ZK and SAUCR are significantly faster. On HDDs, they are about 100× faster. On SSDs, however,



(a) HDD (cluster-1)



(b) SSD (cluster-2)

Figure 8: **Macro-benchmarks.** *The figures show the throughput under various YCSB workloads for memory-durable ZK,* SAUCR*, and disk-durable ZK for eight clients. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

the performance gap is less pronounced. For instance, with a single client, memory-durable ZK and SAUCR are only about 2.1× faster than disk-durable ZK. We found that this inefficiency arises because of software overheads in ZooKeeper's implementation that become dominant atop SSDs.

5.2.2 YCSB Workloads

We now compare the performance of SAUCR against memory-durable ZK and disk-durable ZK across the following six YCSB [23] workloads: load (all writes), A (w:50%, r:50%), B (w:5%, r:95%), C (only reads), D (read latest, w:5%, r:95%), F (read-modify-write, w:50%, r:50%). We use 1KB requests.

Figure 8(a) and (b) show the results on HDDs and SSDs, respectively. For all workloads, SAUCR closely matches the performance of memory-durable ZK; again, the small overheads are a result of writing to one additional node. For write-heavy workloads (load, A, F), SAUCR's performance overheads are within 4% to 9% of memory-durable ZK. For such workloads, memory-durable ZK and SAUCR perform notably better than disk-durable ZK (about 100× and 2.5× faster on HDDs and SSDs, respectively). For workloads that perform mostly reads (B and D), SAUCR's overheads are within 1% to 4% of memory-durable ZK. For such read-heavy workloads, memory-durable ZK and SAUCR are about 25× and 40% faster than disk-durable ZK on HDDs and SSDs, respectively. For the read-only workload (C), all three systems perform the same on both HDDs and SSDs because reads are served only from memory.

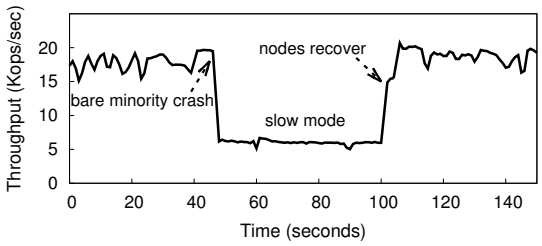


Figure 9: **Performance Under Failures.** *The figure shows* SAUCR*'s performance under failures; we conduct this experiment with eight clients running an update-only workload on SSDs.*

5.2.3 Performance Under Failures

In all our previous performance experiments, we showed how SAUCR performs in its fast mode (without failures). When failures arise and if only a bare majority of nodes are alive, SAUCR switches to the slow mode until enough nodes recover. Figure 9 depicts how SAUCR detects failures and switches to slow mode when failures arise. However, when enough nodes recover from the failure, SAUCR switches back to fast mode.

5.3 Heartbeat Interval vs. Performance

SAUCR uses heartbeats to detect failures. We now examine how varying the heartbeat interval affects workload performance. Intuitively, short and aggressive intervals would enable quick detection but lead to worse performance. Short intervals may degrade performance for two reasons: first, the system would load the network with more packets; second, the SAUCR nodes would consider a node as failed upon a missing heartbeat/response when the node was merely slow and thus react spuriously by flushing to disk or switching to slow mode.

To tackle the first problem, when replication requests are flowing actively, SAUCR treats the requests themselves as heartbeats; further, we noticed that even when the heartbeat interval is lower than a typical replication-request latency, the additional packets do not affect the workload performance significantly. The second problem of spurious reactions can affect performance.

For the purpose of this experiment, we vary the heartbeat interval from a small (and unrealistic) value such as 1 μs to a large value of 1 second. We measure three metrics: throughput, the number of requests committed in slow mode (caused by the leader suspecting follower failures), and the number of flushes issued by a follower (caused by followers suspecting a leader failure). Figure 10 shows the result. As shown, when the interval is equal to or greater than 1 ms, the workload performance remains mostly unaffected. As expected, with such reasonably large intervals, even if the nodes are slow occasionally, the likelihood that a node will not receive a heartbeat or a response is low; thus, the nodes do not react spuriously most of the times. As a result, only a few spurious flushes are issued by the followers, and very few

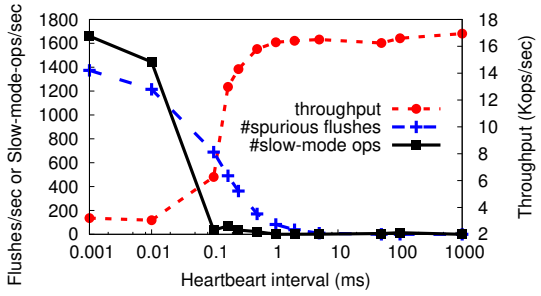


Figure 10: **Heartbeat Interval vs. Performance.** *The figure shows how varying the heartbeat interval affects performance. The left y-axis shows the average number of flushes issued by a follower per second or the average number of requests committed in slow mode by the leader per second. We measure the performance (right y-axis) by varying the heartbeat interval (x-axis). We conduct this experiment with eight clients running the YCSB-load workload on SSDs.*

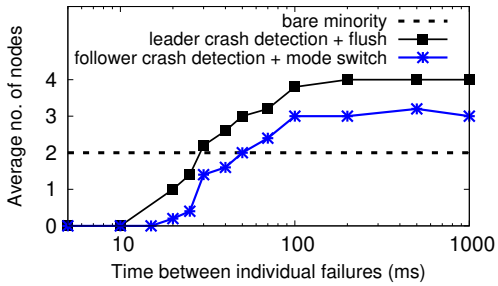


Figure 11: **Correlated Failure Reaction.** *The figure shows how quickly* SAUCR *reacts to correlated failures; the y-axis denotes the number of nodes that detect and flush to disk before all nodes crash when we vary the time between the individual failures (x-axis). We conduct this experiment on SSDs.*

requests are committed in slow mode. In contrast, when the interval is less than 1 ms, the SAUCR nodes react more aggressively, flushing more often and committing many requests in slow mode, affecting performance. In summary, for realistic intervals of a few tens of milliseconds (used in other systems [28]) or even for intervals as low as 1 ms, workload performance remains unaffected.

Finally, although the nodes react aggressively (with short intervals), they do not declare a node as failed because there are no actual failures in this experiment. As a result, we observe that the leader does not step down and the followers do not run for an election.

5.4 Correlated Failure Reaction

We now test how quickly SAUCR detects and reacts to a correlated failure that crashes all the nodes. On such a failure, if at least a bare minority of nodes flush the data to disks before all nodes crash, SAUCR will be able to provide availability and durability when the nodes later recover. For this experiment, we use a heartbeat interval value of 50 ms. We conduct this experiment on a five-node cluster in two ways.

First, we crash the active leader and then successively crash all the followers. We vary the time between the individual failures and observe how many followers detect and flush to disk before all nodes crash. For each failure-gap time, we run the experiment five times and report the average number of nodes that safely flush to disk. Figure 11 shows the result: if the time between the failures is greater than 30 ms, then at least a bare minority of followers always successfully flush the data, ensuring availability and durability.

Second, we crash the followers, one after the other. In this case, the leader detects the failures and switches to slow mode. As shown in the figure, if the time between the individual failures is greater than 50 ms, the system will be available and durable after recovery. As we discussed earlier (§2.3), in a real deployment, the time between individual failures is almost always greater than 50 ms; therefore, in such cases, with a heartbeat interval of 50 ms, SAUCR will always remain safe.

Note that we run this experiment with a 50-ms heartbeat interval; shorter intervals (such as 1 ms used in Figure 10) will enable the system to remain durable and available (i.e., a bare minority or more nodes would safely flush or switch to slow mode) even when the failures are only a few milliseconds apart.

6 Discussion

We now discuss two concerns related to SAUCR's adoption in practice. First, we examine whether SAUCR will offer benefits with the advent of faster storage devices such as non-volatile memory (NVM). Second, we discuss whether applications will be tolerant of having low throughput when SAUCR operates in slow mode.

Faster Storage Devices. The reliability of memory-durable approaches can be significantly improved if every update is forced to disk. However, on HDDs or SSDs, the overhead of such synchronous persistence is prohibitively expensive. New storage devices such as NVMe-SSDs and NVM have the potential to reduce the cost of persistence and thus improve reliability with low overheads. However, even with the advent of such faster storage, we believe SAUCR has benefits for two reasons.

First, although NVMe-SSDs are faster than HDDs and SSDs, they are not as fast as DRAM. For example, a write takes 30 μs on Micron NVMe-SSDs which is two orders of magnitude slower than DRAM [19] and thus SAUCR will have performance benefits compared to NVMe-SSDs. While NVM and DRAM exhibit the same latencies for reads, NVM writes are more expensive (roughly by a factor of 5) [40, 67]. Further, writing a few tens of kilobytes (as a storage system would) will be slower than published numbers that mostly deal with writing cachelines. Hence, even with NVMs, SAUCR will demonstrate benefit.

Second, and more importantly, given the ubiquity of DRAM and their lower latencies, many current systems

and deployments choose to run memory-only clusters for performance [17, 43], and we believe this trend is likely to continue. SAUCR would increase the durability and availability of such non-durable deployments significantly without affecting their performance at no additional cost (i.e., upgrading to new hardware).

Low Performance in Slow Mode. Another practical concern regarding SAUCR's use in real deployments is that of the low performance that applications may experience in slow mode. While SAUCR provides low performance in slow mode, we note that this trade-off is a significant improvement over other existing methods that can either lead to permanent unavailability or lose data. Further, in a shared-storage setting, we believe many applications with varying performance demands will coexist. While requests from a few latency-sensitive applications may time out, SAUCR allows other applications to make progress without any hindrance. Furthermore, in slow mode, only update requests pay the performance penalty, while most read operations can be served without any overheads (i.e., at about the same latency as in the fast mode). Finally, this problem can be alleviated with a slightly modified system that can be reconfigured to include standby nodes when in slow mode for a prolonged time. Such reconfiguration would enable the system to transition out of the slow mode quickly. We believe this extension could be an avenue for future work.

7 Related Work

We now discuss how prior systems and research efforts relate to various aspects of our work.

Situation-Aware Updates. The general idea of dynamically transitioning systems between different modes is common in real-time systems [15]. Similarly, the idea of fault-detection-triggered mode changes has been used in cyber-physical distributed systems [18]. However, we do not know of any previous work that dynamically adapts a distributed update protocol to the current situation. Many practical systems statically define whether updates will be flushed to disk or not [8, 22, 27, 62]. A few systems, such as MongoDB, provide options to specify the durability of a particular request [51]. However such dynamicity of whether the request will be persisted or buffered is purely client-driven: the storage system does not automatically make any such decisions, depending on the current failures.

Recovery. RAMCloud [58,64] has a similar flavor to our work. However, the masters always construct their data from remote backups, unlike SAUCR, which performs mode-specific recovery. SAUCR's recovery is similar to VR's recovery [48]. However, SAUCR's recovery differs from that of VR in two ways. First, in VR, a recovering node waits for a majority responses before it moves to the recovered state, while in SAUCR, a recovering node has to wait only for a bare minority responses. Second, and more importantly, in VR, a responding node can readily be in the recovered state only if it has not yet crashed. In contrast, in SAUCR, a node can readily be in the recovered state in two ways: either it could have operated in fast mode and not failed yet, or it might have operated in slow mode previously or flushed to disk. These differences improves SAUCR's availability. SAUCR's recovery is also similar to how replicated-state-machine (RSM) systems recover corrupted data from copies [1].

Performance Optimizations in RSM systems. Several prior efforts have optimized majority-based RSM systems by exploiting network properties [47, 61]; other optimizations have also been proposed [41, 52]. However, to our knowledge, most of these systems are only memory-durable. SAUCR can augment such systems to provide stronger guarantees while not compromising on performance. A few systems [14, 21, 56] realize that synchronous disk writes are a major bottleneck; these systems have proposed techniques (e.g., batching) that make disk I/O efficient. SAUCR's implementation includes such optimizations in its slow mode.

8 Conclusion

Fault-tolerant replication protocols are the foundation upon which many data-center systems and applications are built. Such a foundation needs to perform well, yet also provide a high level of reliability. However, existing approaches either suffer from low performance or can lead to poor durability and availability. In this paper, we have presented situation-aware updates and crash recovery (SAUCR), a new approach to replication within a distributed system. SAUCR reacts to failures and adapts to current conditions, improving durability and availability while maintaining high performance. We believe such a situation-aware distributed update and recovery protocol can serve as a better foundation upon which reliable and performant systems can be built.

Acknowledgments

We thank Andreas Haeberlen (our shepherd) and the OSDI reviewers for their thoughtful suggestions that improved the presentation of the content significantly. We thank the members of ADSL for their valuable feedback. We also thank CloudLab [63] for providing a great environment for running our experiments. This material was supported by funding from NSF grants CNS-1421033, CNS-1218405, and CNS-1838733, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, NetApp, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-Aware Recovery for Consensus-Based Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[3] Amazon Elastic Compute Cloud. Regions and Availability Zones. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[4] Apache. ZooKeeper. `https://zookeeper.apache.org/`.

[5] Apache. ZooKeeper Configuration Parameters. `https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_configuration`.

[6] Apache. ZooKeeper Leader Activation. `https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_leaderElection`.

[7] Apache Accumulo Users. Setting ZooKeeper forceSync=no. `http://apache-accumulo.1065345.n5.nabble.com/setting-zookeeper-forceSync-no-td7758.html`.

[8] Apache Cassandra. Cassandra Wiki: Durability. `https://wiki.apache.org/cassandra/Durability`.

[9] Apache ZooKeeper. QuorumPacket Class. `http://people.apache.org/~larsgeorge/zookeeper-1075002/build/docs/dev-api/org/apache/zookeeper/server/quorum/QuorumPacket.html`.

[10] Apache ZooKeeper. ZooKeeper Overview. `https://zookeeper.apache.org/doc/r3.5.0-alpha/zookeeperOver.html`.

[11] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.0 edition, May 2015.

[12] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems. Technical Report CMU-CS-02-129, School of computer science, Carnegie-Mellon University, 2002.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[14] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[15] Alan Burns. System Mode Changes - General and Criticality-Based. In *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8, 2014.

[16] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[17] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[18] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault Tolerance and the Five-second Rule. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

[19] Chris Mellor. Storage with the speed of memory? XPoint, XPoint, that's our plan. `https://www.theregister.co.uk/2016/04/21/storage_approaches_memory_speed_with_xpoint_and_storageclass_memory/`.

[20] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data

Loss in Cloud Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, CA, June 2013.

[21] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[22] CockroachDB. CockroachDB Cluster Settings. `https://www.cockroachlabs.com/docs/stable/cluster-settings.html`.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[24] CoreOS. etcd Guarantees. `https://coreos.com/blog/etcd-v230.html`.

[25] DataCenterKnowledge. Lightning Disrupts Google Cloud Services. `http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikes-google-data-center-disrupts-cloud-services/`.

[26] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[27] Elasticsearch. Translog Settings. `https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html#_translog_settings`.

[28] Etcd. Etcd Tuning. `https://coreos.com/etcd/docs/latest/tuning.html`.

[29] Flavio Junqueira. Transaction Logs and Snapshots. `https://mail-archives.apache.org/mod_mbox/zookeeper-user/201504.mbox/%3CDA045626-54A4-4F8A-96C0-69DA574D9807@yahoo.com%3E`.

[30] Flavio Junqueira. [ZooKeeper-user] forceSync=no. `http://grokbase.com/p/zookeeper/user/126g0063x4/forcesync-no`.

[31] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[32] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[33] Google. Google Cluster Data. `https://github.com/google/cluster-data`.

[34] Google Code University. Introduction to Distributed System Design. `http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html`.

[35] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[36] Henry Robinson. Consensus Protocols: Paxos. `http://the-paper-trail.org/blog/consensus-protocols-paxos/`.

[37] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, MA, June 2010.

[38] Jay Kreps. Using forceSync=no in Zookeeper. `https://twitter.com/jaykreps/status/363720100332843008`.

[39] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance Broadcast for Primary-backup Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[40] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (USENIX '18)*, Boston, MA, July 2018.

[41] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[42] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[43] Ken Birman. What we can learn about specifications from ZooKeeper's asynchronous mode, and its unsafe ForceSync=no option? `http://thinkingaboutdistributedsystems.blogspot.com/2017/09/what-we-can-learn-from-zookeepers.html`.

[44] Madhukar Korupolu and Rajmohan Rajaraman. Robust and Probabilistic Failure-Aware Placement. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 213–224. ACM, 2016.

[45] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[46] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[47] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[48] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.

[49] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[50] Microsoft Azure. Azure Availability Sets. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/tutorial-availability-sets#availability-set-overview`.

[51] MongoDB. MongoDB Write Concern. `https://docs.mongodb.com/manual/reference/write-concern/`.

[52] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[53] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[54] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[55] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, ON, Canada, August 1988.

[56] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[57] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[58] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[59] Parsely Inc. Streamparse: Configuring Zookeeper with forceSync = no. `https://github.com/Parsely/streamparse/issues/168`.

[60] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[61] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[62] RethinkDB. RethinkDB Settings - Durability. `https://rethinkdb.com/docs/consistency/`.

[63] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[64] Ryan Scott Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford University, 2013.

[65] UWSysLab. VR Implementation. `https://github.com/UWSysLab/NOPaxos/tree/master/vr`.

[66] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 362–367. IEEE, 2002.

[67] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[68] Zookeeper User Mailing List. Unavailability Issues due to Setting forceSync=no in ZooKeeper. `http://zookeeper-user.578899.n2.nabble.com/forceSync-no-td7577568.html`.

Taming Performance Variability

Aleksander Maricq[*] Dmitry Duplyakin[*] Ivo Jimenez[†] Carlos Maltzahn[†]
Ryan Stutsman[*] Robert Ricci[*]

[*]*University of Utah,* [†]*University of California Santa Cruz*

The performance of compute hardware varies: software run repeatedly on the same server (or a different server with supposedly identical parts) can produce performance results that differ with each execution. This variation has important effects on the reproducibility of systems research and ability to quantitatively compare the performance of different systems. It also has implications for commercial computing, where agreements are often made conditioned on meeting specific performance targets.

Over a period of 10 months, we conducted a large-scale study capturing nearly 900,000 data points from 835 servers. We examine this data from two perspectives: that of a service provider wishing to offer a consistent environment, and that of a systems researcher who must understand how variability impacts experimental results. From this examination, we draw a number of lessons about the types and magnitudes of performance variability and the effects on confidence in experiment results. We also create a statistical model that can be used to understand how representative an individual server is of the general population. The full dataset and our analysis tools are publicly available, and we have built a system to interactively explore the data and make recommendations for experiment parameters based on statistical analysis of historical data.

1 Introduction

Variability is an unavoidable aspect of computer systems performance. In the research community, rigorous comparison of systems requires understanding, analysis, and control of system variability [45, 21, 12, 27]. In the commercial space, understanding and controlling performance variability is critical to providing good user experience [14, 23] and to plan resource provisioning [1].

Large systems have many sources of performance variability (hereafter referred to as simply "variability"), but one that cannot be avoided is the variability of hardware. For this paper, we consider two types of variability: variability of the *same physical system under repeated experiments*, and variance between different physical systems that are supposedly *identical*. Hardware can exhibit variability due to temperature [17], variations in timings and orderings, remapped storage blocks [44] or memory cells [52], variance in manufacture [65], "fail-slow" hardware [25], and many more causes.

We present findings and recommend best practices from two different perspectives: infrastructure-as-a-service (IaaS) *providers* and their *users*. On the provider side, we consider the amount of variability that can reasonably be controlled by factoring out unrepresentative servers, and how to reliably detect such devices. On the user side, we consider the variability that remains, how to cope with it when running experiments, and how to avoid certain pitfalls. Our intention is to make experimentation in the face of variability easier by demystifying its sources and quantities and by making concrete recommendations.

We collected data from servers in CloudLab [60], a platform for systems research that provides exclusive "raw" access to compute and storage resources. CloudLab allocates an entire server to one user at a time; we ran our benchmarks on servers when they were not allocated to any other user. This enables us to report performance numbers that users could reasonably expect to see in their own applications, unaffected by other simultaneous users. This data was collected on an IaaS provider that constitutes research infrastructure (a "testbed"), but we believe these lessons also apply to other settings in which there is an agreement between providers and users to supply a specific, measurable level of performance, such as clouds and datacenters.

In this paper, we:

- Provide a refresher of statistical methods used to assess confidence in performance results (§2) and the impact of variability on experimentation
- Describe our testing framework, the servers we tested, and the resulting dataset (§3)
- Analyze this dataset (§4) to understand the sources and quantities of variability
- Present a new method for estimating how many repetitions of an experiment to run (§5) and CONFIRM, our tool to aid experimenters in gathering statistically significant results
- Devise methods for service providers to identify servers with unrepresentative behavior (§6).
- Cover defensive practices (§7) that help avoid pitfalls with respect to variability

Throughout, we identify specific findings (identified with ◊) aimed at helping service providers provide more

consistent facilities and assisting users in understanding and coping with the variability inherent in computer systems. We close with related work and future directions.

2 The Statistics of Performance Variability

The fundamental way variability impacts systems research is that it affects our *confidence* in the statistical power of our results and the correctness of conclusions that we draw. When we run experiments and calculate statistics (mean, median, etc.) we are producing *empirical* statistics from a sample (a finite number) of a notional *population* (an infinite number) of executions. As we run more *repetitions* of an experiment, we can be more confident that our empirical distributions are close to the population distributions, and for key statistics such as the mean and median, we can compute *confidence intervals* (CIs).

For a chosen *confidence level* α, a CI defines a range in which we are α% sure that the population mean lies. For example, a sample mean of 10.0, with a CI of $9.9-10.1$ at 95% confidence indicates a 95% confidence that the true mean lies within $r = 1\%$ of our estimate 10.0. In order to make a strong statement that one sample mean is higher than another, their CIs should not overlap [31]; if they do, it is possible that the true population means have the reverse relationship. When an experiment is analyzing a small effect (for example, a 5% performance improvement), a wide CI may invalidate the conclusion.

> ◊ ***Perform enough repetitions to achieve tight confidence intervals***
> Techniques from statistics provide robust foundations for making strong statements about performance differences between systems.

Statistical methods fit into two broad classes: parametric and nonparametric techniques. The former class, which is more well-known, relies on the assumption that the analyzed data stems from known probability distributions, typically the Normal/Gaussian distribution. A variety of closed-form expressions for statistics of interest enable powerful parametric analysis. In contrast, nonparametric techniques are used when the probability distributions are *unknown*, and require fewer assumptions. Nonparametric methods, which have fewer closed-form equations, involve less powerful counterparts of popular parametric techniques, e.g. the Kruskal-Wallis test [40] instead of ANOVA. In nonparametric analysis, empirical mean and standard deviation can be computed, but their interpretation is different compared to the parametric case: rather then using them to fit distribution curves, they reveal only high-level information about the shapes of population distributions. The two most common metrics of interest, the median and CI for the median, can be used to compare pairs or sets of sampled nonparametric distributions.

Many studies suggest that the normality assumption does not hold for the data obtained in computer systems experiments, especially when the data includes measurements of performance. This applies both on a single machine [34] and in parallel programs running on supercomputers [67]. Indeed, as we document in §4.3, most data in our dataset does not follow the normal distribution. Thus, we adopt nonparametric statistics for the remainder of this paper, and recommend that, for performance experiments, these methods be used unless normality can be demonstrated. In [27] and [13], the authors provide advice for statistically sound performance analysis and argue for applying robust nonparametric techniques.

In nonparametric analysis, one uses the median, rather than the mean, as the measure of central tendency. To get CIs for a set of measurements X, one first sorts X. Then (as described in [41]), compute $\left\lfloor \frac{n-z\sqrt{n}}{2} \right\rfloor$ and $\left\lceil 1+\frac{n+z\sqrt{n}}{2} \right\rceil$, where n is the number of elements in X, and z is the *standard score* (or *z-score*) [31]. z depends only on the desired confidence level, and is 1.96 for the commonly-used level of $\alpha = 95\%$. These two numbers are then used as indexes into the sorted X: the values at those locations are the top and the bottom bounds of the CI. Note that one of these numbers will be larger than the median (at index $\lfloor \frac{n}{2} \rfloor$) and the other will be smaller, and they will not necessarily be symmetric around the median. These bounds tend to get tighter—to approach the sample mean—with more repetitions. Typically, we are concerned with the relative difference r% between the CI bounds and the mean.

A natural question is how many repetitions of an experiment are likely to be needed to achieve a sufficiently narrow CI (e.g., indicating that the empirical median differs from the true median by no more than $r = 1\%$) for a given confidence level α (e.g. 95%): we want to be sure to run enough repetitions to be confident in our results, but don't want to waste time running more than necessary. We use $E(r,\alpha,X)$ to represent this value for a set of experiment results X. The value of E can vary widely depending on the data in X; intuitively, the more variation between measurements in X, the more runs it will tend to take to narrow the CI to the target of r%. So that we can compare values of E to each other, for the remainder of this paper we adopt $E(1\%,95\%,X)$ as our standard target and denote it as $\check{E}(X)$. It is important to note that this is an *estimate* of what is required to get the desired confidence: empirical CIs must still be computed from the data gathered.

Finding $\check{E}(X)$ for parametric models is straightforward, as most such models have a closed-form equation that uses an estimate of the variance of X, obtained by running

a handful of exploratory experiments. In the nonparametric case, this number is harder to find, since we cannot make any assumptions about the distribution and there is therefore no equation we can use. One of the major contributions of this paper, covered in §5 is a resampling-based technique for estimating $\check{E}(X)$ for nonparametric models, and a tool we have built that makes it easy for experimenters to get these estimates.

3 Methodology

Over a period of 10 months, from May 20, 2017 to April 1, 2018, we collected performance measurements on servers that are part of the three CloudLab [60] clusters. Our experiments were run while servers were *not* allocated to other users, meaning that they did not affect, nor were they affected by, other users of the facility.

3.1 Testing Framework

Our testing framework is built with `geni-lib` [5], a Python library for interacting with GENI-compatible testbeds such as CloudLab. We wrote an orchestration script which selects free servers, runs benchmarks, and collects the results. In order to avoid consuming excessive resources on CloudLab, this script runs at a fixed interval every six to eight hours on each CloudLab cluster. Three to five servers (depending on the size of the cluster) are selected by fetching a list of the target cluster's available servers, checking them against our database of previous runs, and prioritizing never-tested servers, followed by least recently tested servers. Servers that have had a recent failure are not re-tested for a week to avoid having them remain at the highest priority.

Once the test servers are provisioned, the orchestration script waits for the provisioning process to be completed, logs into the server, and automatically runs the tests (described below). A single run can take between 30 minutes and 5 hours; the majority of this time is spent running disk tests.

As a side effect of the way that the CloudLab allocation policies and usage patterns work, servers were not sampled uniformly: some servers were unavailable for up to months at a time, as they were part of long-running experiments. In general, the more popular the type of server, the more sparsely sampled it is. Times of heavy testbed utilization, such as major deadlines, are also sparsely sampled. This requires us to use analyses that are robust with respect to different sample sizes.

3.2 Benchmarks

We selected a set of benchmarks to cover three key resources: memory, storage, and networking. In our selection of benchmarks, we struck a balance between observing the performance of the hardware when pushed to the limit (to detect degraded performance), and what might be seen in a more typical application (to understand "typical" behavior of the hardware). Hyper-optimized benchmarks can often come at the expense of practicality, and often make use of instructions, settings, and "tricks" that are limited to specific processors or I/O devices. We also required benchmarks that were portable across different architectures, due to the presence of both `x86-64` and `ARM` machines in CloudLab. Our primary benchmarks follow both principles, and we have some supplementary x86-specific benchmarks that use intrinsics to maximize performance. Memory and storage results have been collected since the beginning of our study, and we started collecting network benchmarks about 6 months later.

Memory We use two different benchmark suites for our memory tests. First, STREAM [43] (a standard benchmark for HPC machines) gathers a simple set of single-threaded and multi-threaded micro-benchmarks that perform basic operations such as memory copies and simple mathematical manipulation of memory contents. Second, we use a suite of micro-benchmarks by Alex W. Reece [51, 50] for supplemental non-portable tests utilizing Intel x86 intrinsics such as SSE and AVX. We found that, while absolute numbers differed, these other benchmarks did not alter our conclusions, so we discuss only the STREAM benchmarks in this paper. All tests use sufficient memory to minimize caching effects.

While we made no modifications to any timed portions of the benchmarks, we did modify both benchmarks to provide more complete reporting of statistics at the end of their runs. In addition, we altered the overall STREAM workflow to run a single-threaded test followed by a multi-threaded test. In the case of Intel processors, we run tests both with a standard frequency-scaling setting and with a setting that disables turbo boost and sets the performance governor to "performance." In the case of multi-socket machines, we test on each socket independently using `numactl` to avoid bottlenecks with QPI. Both memory benchmarks are built from source during each run using `gcc` with exactly the same compile flags every time; this helps with our multi-architecture environment, and means that `gcc` applies the optimizations appropriate to that environment.

Storage We test storage by using `fio` [3] to issue direct 4KB asynchronous I/O requests to target raw block devices. For the boot device, we run `fio` on the partition of that device containing the remaining empty space. Otherwise, we run `fio` on the entire device. We test both sequential and random reads and writes independently, and each workload is run both with a high and low number of I/Os issued to the device at any given time. A low I/O depth (we use 1) is more sensitive to device latency, whereas a high one (we use 4096) is more sensitive to bandwidth and internal parallelism. In the case of SSDs,

Type	#	Model	Processor	S	C	RAM	Boot Disk	Other Disks
m400	315	HPE m400	ARM64 X-Gene	1	8	64 GB (8x4)	SATA III SSD	None
m510	270	HPE m510	Xeon D-1548	1	8	64 GB (4x8)	NVMe SSD	None
c220g1	90	Cisco c220m4	Xeon E5-2630v3	2	16	128 GB (8x8)	SAS-2 HDD	SAS-2 HDD &
c220g2	163	Cisco c220m4	Xeon E5-2660v3	2	20	160 GB (8x10)	SAS-2 HDD	SATA III SSD
c8220	96	Dell C8220	Xeon E5-2660v2	2	20	256 GB (16x16)	SATA II HDD	SATA II HDD
c6320	84	Dell C6320	Xeon E5-2683v3	2	28	256 GB (16x16)	SATA II HDD	SATA II HDD

Table 1: Server configurations. "S" is the number of sockets, and "C" is the total core count (across all sockets). RAM is described as "(DIMM size x # DIMMs)". SAS-2 HDDs are all 10k RPM, and SATA II HDDs are all 7.2k RPM.

we issue a TRIM to the device using `blkdiscard` before we run any write workload. This clears certain block state, allowing for more efficient write operations [26]. We install `fio` from the Ubuntu package repository.

Network For each site, we set a fixed destination server that every server runs network tests against over a shared VLAN. For latency tests, we use a simple ICMP `ping` in flood mode. For Bandwidth tests, we use `iperf3` [30] with TCP and take measurements bidirectionally. Some of servers we test are rack-local with the destination server, and others require multiple layer-2 hops. Since CloudLab makes its topology public, we know that all non-local servers we are testing are three to four Ethernet hops away, and we record switch-path information along with each test. We install `iperf3` from the Ubuntu package repository, and `ping` is already bundled with the base operating system.

3.3 Servers Tested

We gathered our results from CloudLab's three primary clusters: Utah, Wisconsin, and Clemson. Servers at each site are divided into a small number of distinct homogeneous *types*; no sites currently have overlapping types. All servers we tested are interconnected via a 10 Gbps "experiment" network within each site. At the time of our tests, each of these sites had two "dominant" types consisting of tens to hundreds of servers. Some sites have types with only a few instances containing specialized hardware such as GPUs or many disks; we did not test these types to avoid consuming CloudLab's scarcest resources.

A summary of the server types we tested can be found in Table 1. The two Utah types are designed on the low-power and high-density Moonshot platform from HPE, with 45 servers in each 4U chassis. The two Clemson types are somewhat less dense, with four to eight servers per 2U chassis, while the Wisconsin servers are in independent 1U chassis. The Wisconsin servers have the most disks, with each server having a boot HDD, plus one "extra" HDD and SSD each. More detailed information regarding the experimented-upon server types, such as specific component models, can be found on the CloudLab Hardware documentation pages [61, 59].

3.4 Software Consistency

While we focus on hardware-based variance in this paper, we recognize that software differences can have a major impact on performance. To this end, our testing framework tracks, for each test, the version information of the kernel, versions of key packages (such as the compiler), and the revision of our repository containing our test script and memory benchmark sources. The key software remained at the same version throughout this test: the operating system release (Ubuntu 16.04, standard CloudLab image), the Linux kernel release (`4.4.0-75-generic`), `ping` (`iputils-s20121221`), and `iperf3` (`3.0.11`). While our testing repository was updated several times over the testing period, no modifications were made to any timed areas of our memory benchmarks. Finally, almost all runs utilized the same `gcc` version (`5.4.0`) and `fio` version (`2.2.10`). A very small percentage ($< 1\%$) of our runs used slightly earlier versions of both `gcc` and `fio`, so to maintain software consistency we excluded them while performing our analysis.

CloudLab released disk images with mitigations for Spectre and Meltdown (which are known to affect performance) on April 2, 2018; we intentionally use data through April 1 so that we can focus on hardware variance in this paper. We are continuing to collect data, and expect variance due to system software to be an interesting topic for study in its own right.

3.5 Resulting Dataset

From the period of May 20th 2017 to April 1st 2018, we collected 10,400 total runs from 835 total machines. A complete breakdown of machines tested and runs by hardware type can be found in Table 2. Since each run involved execution of a multitude of benchmarks in different configurations, we ended up with a total of 892,964 distinct data points over this period.

We use the term "configuration" to refer to the combination of hardware type, configuration, and benchmark settings. For example, the possible memory configurations come from varying hardware type, socket number, single- or multi-threaded operation, frequency scaling, and type of memory operation; this results in 590 possible configurations for memory. Similarly, there are

CloudLab Site	Hardware Type	Tested/ Total Servers	Total Runs	Mean/ Median Runs
Utah	m400	223/315	3583	16/8
	m510	221/270	2007	9/7
Wisconsin	c220g1	88/90	800	9/7
	c220g2	125/163	1527	12/8
Clemson	c8220	96/96	1742	18/12
	c6320	82/84	741	9/8
Total		835/1,018	10,400	12/8

Table 2: Coverage of our dataset

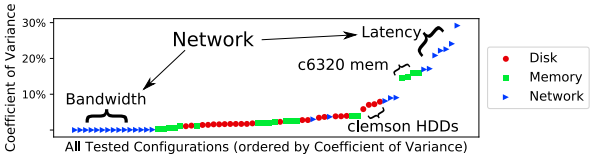


Figure 1: CoV for a variety of configurations.

96 possible configurations for storage, and 27 possible configurations for network tests. Each data point in the dataset comes from executing one configuration.

4 Understanding Variability

We begin our analysis of the dataset with an exploration of some of the key statistics computed from it. All data used in this section has had measurements from servers that are outliers removed, so it represents the unavoidable variation that experimenters must cope with even when the service provider does its best to provide consistently-performing servers. The procedure that we developed for removing unrepresentative servers is described in §6.

4.1 Unavoidable Variability

We first use our dataset to answer questions of importance to experimenters and other users who need consistency from the platforms they use. These stem from the basic question "How much variability must I account for in my experiments?"

Aiming to perform fair high-level assessment, we select a subset of 70 benchmark × hardware combinations with relatively even distribution: 24 disk (all for boot devices), 19 memory (variants of *copy* benchmark), and 27 network (both latency and bandwidth) configurations being tested. We use *coefficient of variance* (CoV), the ratio of the standard deviation to the mean, to compare these configurations; absolute *standard deviation* cannot be used here due to the difference in the scales and units of compared measurements. Displayed in Figure 1 and ordered by CoV, the analyzed configurations reveal the following insights:

Networking Both top and the bottom of the list are dominated by the network tests: primarily, latency tests are at the top with CoV in the range $[16.9\%, 29.2\%]$, while the bandwidth tests are at the bottom with CoV $< 0.1\%$. For the configuration with the largest CoV, we notice that the standard deviation is $7.7\mu s$ is quite small in absolute terms. However, it is a significant fraction of the empirical mean for the latency at $26.3\mu s$. This seems to stem from a couple of sources: First, we are using standard, unoptimized, tools to measure latency, and the timescales are small enough that effects within the kernel networking stack are noticeable (even loopback `ping` displays some variation). Second, the granularity of timestamps reported by `ping` is sufficiently coarse ($1\mu s$) that measurements group into discrete bands instead of being continuously distributed. In contrast, the 3.3×10^5 standard deviation on most bandwidth tests corresponds to only 330kbps out of the median of 9.4Gbps. We note that CloudLab allocates network bandwidth in such a way as to attempt to guarantee each experiment the full bandwidth it has requested, free of interference from other users. The low CoV in bandwidth tests suggests that it is effective in doing so. Not all datacenters have similar bandwidth allocation policies, and this result may vary in a different setting.

c6320 Memory A block of memory tests for the `c6320` servers stands out for having higher CoVs than other memory tests: they are in the range $[14.5\%, 16.0\%]$. There is no clear cause for this variability, but it is remarkable for being the only set of configurations for which a particular type of server is tightly grouped. The lesson we take from this is that it it underscores the need to test all configurations rather than assuming that similar types of resources exhibit similar variability.

Clemson HDDs Clemson servers stood out in another way as well: the HDDs on both Clemson types show moderately high CoV for high-iodepth random I/O for both reads and writes. These disks (which are the same model on both types) are the only 7.2k RPM HDDs in CloudLab, as well as the only SATA HDDs.

Bulk of the Tests The remaining block of 44 tests consists of intermingled disk and memory configurations. CoVs for this set of tests are in the range $[0.3\%, 9.0\%]$. While this is a fairly wide range, there is no clear pattern within it; for example, the data does not support the hypothesis that disk bandwidth consistently exhibits more variability than memory or vice versa. Also, unlike the results for `c6320`, individual server types show no grouping, leading us to the conclusion that, on the whole, there is little correlation between server type and CoV. Based on this analysis, we expect CoV for hardware performance metrics observed in practice to be roughly in this range most of the time and, in rare cases, exceed the 10% mark.

HDDs@c8220	HDDs@c220g1	SSDs@c220g1
6.85% (rr, H)	5.66% (r, L)	9.86% (rr, L)
6.42% (rw, H)	3.68% (rr, H)	5.38% (r, L)
6.08% (rr, L)	1.93% (r, H)	4.65% (rw, L)
5.82% (r, L)	1.90% (w, H)	3.95% (w, L)
5.32% (rw, L)	0.99% (rw, L)	1.00% (w, H)
4.96% (w, L)	0.93% (rr, H)	0.68% (r, H)
1.27% (w, H)	0.58% (rr, L)	0.53% (rw, H)
1.20% (r, H)	0.14% (w, L)	0.09% (rr, H)

Table 3: Coefficient of Variance. Values are annotated with the type of test and iodepth: *r*ead, *w*rite, *r*and*r*ead, *r*and*w*rite. "L" and "H" denote iodepth 1 and 4096.

> ◊ ***Some amount of variation is unavoidable***
> Some degree of variation in hardware performance is unavoidable, no matter what steps the facility provider takes to provide consistent hardware. Coefficients of Variance of up to 10% may be attributed to hardware variability and considered expected, while higher values may indicate room for improvement from the measurement standpoint.

For the aforementioned CoV range, we determine that the configuration with CoV $= 0.3\%$ is likely to require only $\check{E}(X) = 10$ experiments in order to make the corresponding CI sufficiently small. In contrast, this number significantly increases, up to $\check{E}(X) = 240$, for the configuration with CoV $= 9.0\%$. This demonstrates the need for careful experiment design that takes into account variation of the specific resources that are exercised, and we present further analysis of the relationship between the two metrics in §5.

4.2 Disk I/O

SSDs are well-known to have complex performance profiles [26] due to the write characteristics of flash and their internal Flash Translation Layers (FTLs). In addition, different types of HDDs have performance characteristics based on their rotational latency, attachment protocol, density, etc. We wanted to answer the question "Are SSDs more consistent (lower CoV) than HDDs?", so we look at the variability for two different types of HDDs (one model at Wisconsin and one at Clemson), and for SSDs at Wisconsin. The devices at Wisconsin are higher-end: 10k RPM SAS-2 HDDs and enterprise Intel SSDs, while the HDDs at Clemson are 7.2k RPM SATA III devices.

As shown in Table 3, the answer to our question depends on the level of parallelism (iodepth) and the type of HDD. With high iodepth, SSDs use their internal parallelism and demonstrate both much higher performance and more consistency. The SSDs we tested are 2.3–2.4

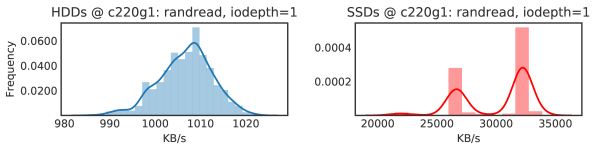


Figure 2: Histogram of iodepth=1 randread on c220g1.

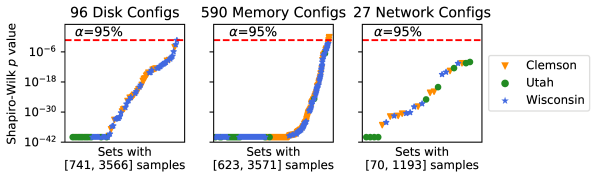


Figure 3: Testing normality of the collected data.

times faster on sequential tests than HDDs, and from 82.5 up to 262.3 times faster on random reads and writes. CoVs for these tests were in the range $[0.09\%, 1.0\%]$ for SSDs, lower than most HDD CoVs.

On HDDs, unsurprisingly, iodepth is not strongly correlated with CoV: these devices have less internal parallelism, and it is harder to exploit due to the lack of an abstraction layer as complex as the FTL. Because SSDs have such high CoV on low-iodepth tests, some HDDs are competitive in terms of CoV (if not absolute performance). The reason for this can be seen in Figure 2, which examines the case of random reads. HDDs have a performance curve that is fairly compact: it is dominated by seek time and rotational delay, and roughly bounded by the maximum values of those two variables. This curve is more compact for the higher-RPM SAS dives at Wisconsin, which have lower CoV for most low-iodepth test than the SSDs. The SSDs that we tested, on the other hand, exhibit a bimodal pattern; the exact underlying cause is difficult to ascertain because of the opaque nature of the vendor's FTL, but the effect on experiments is clear and dramatic. The lower-RPM SATA HDDs at Clemson are less competitive against the SDDs in terms of CoV; this is likely due, in part, to their higher rotational latency.

4.3 Testing For Normality

The statistics commonly used to analyze the performance of computer systems [31] tend to assume that measurements are normally distributed. We use the Shapiro-Wilk test [55] to test for normality in our dataset, and find that benchmarks of individual configurations *across different servers* are **not** normally distributed. We apply this test to all configurations and show our results in Figure 3. Each point, shown in the order of increasing p-values, characterizes samples for a specific configuration. For points above the threshold, we cannot reject the *null hypothesis* (stating that the samples come from populations which have normal distributions). For points below this

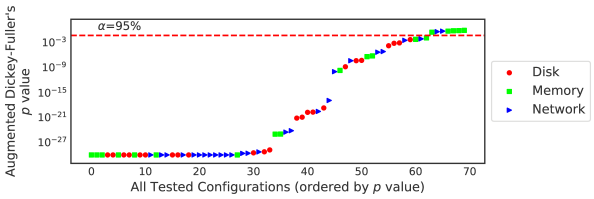


Figure 4: Testing stationarity of the collected data.

threshold, we reject the null hypothesis at this confidence level (95% in this figure), and assume non-normality. Our analysis shows that we should reject the null hypothesis for over 99% of the configurations (710 out of 713). Intuitively (and confirmed by inspecting the underlying data), when we measure maximum bandwidth of a device, there is a practical maximum that cannot be exceeded except by measurement error, and most measurements lie near this maximum. On the other hand, some measurements are significantly lower than the maximum, leading to a skewed distribution with a compressed range above the median and a much larger range below it. The situation is reversed for latency tests. Considering the large number of samples in the analyzed configurations, from 70 in the smallest up to 3,571 in the largest, we reject the normality hypothesis for tests across servers; hence, our focus on nonparametric analyses in this paper.

◊ ***Use nonparametric confidence intervals to avoid assumptions of normality.***
Many computer systems performance results have skewed distributions (longer tails on one side); nonparametric confidence intervals are simple to compute, and work for these distributions (as well as normally-distributed results).

We also test normality for sets of data points that are all drawn from the *same server*. We filter data by selecting servers with at least 20 data points coming from memory tests (this number coming from [55]). Given the way we schedule tests, many servers have not executed more than 20 tests and thus this subset corresponds to 42,680 data points. After applying Shapiro-Wilk to this subset, roughly half of the points (26,695) can be considered to be coming from a normal distribution. Intuitively, we can assume normality in this subset because data points are obtained by running a configuration on the same machine, that is, the hardware and software are the same for all points. This suggests that experimenters should proceed with caution when analyzing results from a single server: data *may* be normally-distributed and thus suitable for analyses that assume normality, but a test such as Shapiro-Wilk should be run to confirm or deny this assumption.

◊ ***For some configurations, single-server tests can be assumed to be from a normal distribution.***
Evaluating normality for tests run on a single server can simplify the analysis since parametric statistics can be employed for these single-server results.

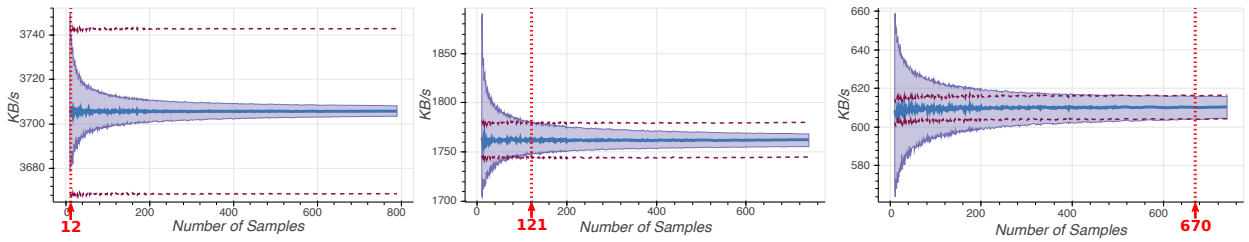
4.4 Checking Stationarity

Most statistical tests—including confidence intervals—assume *stationarity*: that is, that the properties of the underlying distribution (such as median and variance) do not change over time. In addition to affecting data analysis, non-stationary distributions would harm reproducibility: if performance is not stable over time, future experiments cannot reliably be compared to past ones. We use the Augmented Dickey–Fuller (ADF) [15] test to check for stationarity in our data.

For all 70 configurations shown in Figure 1, we run ADF and get a range of p values allowing us to accept or reject the non-stationarity null hypothesis in each case. These values, shown in Figure 4, indicate that nearly all of the analyzed datasets present strong evidence for stationarity: we can reject the hypothesis that they are non-stationary with the confidence level $\alpha = 95\%$ for all points below the line. Among the handful of non-stationary cases (above the line), we find several memory (*copy* benchmark run on `c220g1`) and network bandwidth (also run on `c220g1`) tests. Among the evaluated disk tests there is more tendency towards non-stationarity in the tests with iodepth $= 1$. Recall that our measurements are not sampled from servers uniformly, as described in §3. This appears to be a cause of some of the non-stationary patterns we observe: during some periods, certain servers are over-sampled, and, as they are slightly outside the mean for the whole population, this produces a temporary shift in the mean. These effects could be visible to CloudLab's users, since during periods of heavy utilization, users frequently creating and terminating experiments could see the same set of servers repeatedly. Our remedy to this, detailed in §6, is to find and remove servers that have significant statistical departures from the rest of the population.

5 CONFIRM: How Many Measurements Are Enough?

Given that some amount of variability is inevitable, we turn to a perennial question for experimenters: "How many repetitions do I need to run in order to be confident in my results?" As described in §2, given a set of measurements and a desired confidence level (such as 95%), we can compute a confidence interval (CI) for the mean or median. A standard procedure is to "invert" this calculation, and for a given desired confidence level and CI



(a) 88 HDDs at Wisconsin (c220g1), random reads, iodepth=4096

(b) 82 HDDs at Clemson (c6320), random reads, iodepth=4096

(c) 82 HDDs at Clemson (c6320), random reads, iodepth=1

Figure 5: Nonparametric confidence intervals produced by CONFIRM. As the number of samples grows, 95% CIs (filled areas) for the medians (thick blue lines) shrink and fit within the 1% error bounds (dashed lines). This stopping condition is depicted with red lines and annotated with the numbers of recommended measurements $\check{E}(X)$.

width, estimate how many repetitions are likely necessary to achieve the desired confidence.

When assuming normality, there is a closed-form equation to calculate this estimate [31]; the main input to this equation is an estimate of variance, typically obtained by running a small number of trial runs. In the nonparametric space, there is no closed-form equation, so producing such an estimate requires a more complex technique. We have developed such a technique using *resampling*:

For a set of collected measurements X with n values, we randomly select a subset of $s \leq n$ values for which we estimate the bounds of the CI for the median as described in §2. We shuffle X, select another subset of s values, and obtain new estimates of the CI. After we repeat this process c times, we calculate the means of the lower and upper CI bounds. Obtained using *sampling without replacement*, each of these random selections or "trials" represents a hypothetical scenario where a smaller, partial subset of measurements was collected by an experimenter. The aforementioned averaging eliminates the dependence of the results on the properties of a particular subset and provides an aggregate view on the convergence of the CI observed across many trials. The results presented in the rest of the paper are obtained using $c = 200$. To estimate the recommended number of measurements $\check{E}(X)$, we start at $s = 10$, assuming that smaller subsets are insufficient to estimate nonparametric CIs reliably and should not be considered. Then, we increase s until $s = n$ or the mean CIs fit within the desired error bounds. In the former case, we conclude that these n samples are insufficient for meeting the stopping condition, while in the latter case, we note that the experimentation could have stopped after $\check{E}(X) = s$ measurements according to the selected allowed error and confidence level.

We have implemented this technique in a service we call CONFIRM or CONFIdence-based Repetition Meter. This dashboard imports our benchmarking datasets and facilities interactive nonparametric analysis of CIs for measurements collected from individual servers, groups of servers, and entire hardware types available on CloudLab. We present three analyses here to demonstrate how CONFIRM can help guide experimentation: looking at the how variability affects experiments on different types of HDDs; quantifying how much a single outlier can increase the number of repetitions that must be run; and looking at the relationship between variance and $\check{E}(X)$.

HDD Variation In the first set of experiments, we compare 88 HDDs at Wisconsin with 82 HDDs at Clemson, revisiting results from §4.2 from the perspective of variation. CONFIRM produces visualizations of the CIs and the $\check{E}(X)$ estimates that are depicted in Figure 5. Figure 5 (a)-(b) show the difference of over 10× in $\check{E}(X)$ for two disk types running the same benchmark (random reads, iodepth = 4096), with Clemson disks exhibiting higher variance and wider CIs. A similar benchmark—random reads, iodepth = 1—demonstrates an even more severe case, as shown in Figure 5 (c). For the same set of Clemson HDDs we have to use as many as 670 samples (almost all of the measurements we have collected) in order to fit the CI within the same 1% error bounds. If we were to select a set of servers based on reproducibility of disk-heavy workloads, the Wisconsin servers would be the clear choice; conversely, if our experiments must be run on the Clemson servers, we will need to be careful to run many repetitions to get statistically significant results.

Effects of Outliers In the second set of experiments, we start with a randomly selected set of 9 c220g2 servers at Wisconsin, add one more "badly" performing server of the same type (one that will be eliminated using the method in §6), and analyze CIs for memory tests with and without this outlier. We run CONFIRM on the combination of the selected servers and four variations of the *copy* memory test and record obtained $\check{E}(X)$ estimates in Table 4. We can see that inclusion of this server results in a 2.1–5.9× increase in the recommended number of repetitions. Our analysis shows that the distribution of the performance data obtained on these 10 servers is highly skewed, with the "long tail" caused by the low-

Memory test / frequency-scaling / tested socket	9 servers	10 servers (same 9 + 1 "outlier" server)
copy / no / 0	18	63
copy / no / 1	10	58
copy / yes / 0	33	68
copy / yes / 1	10	54

Table 4: Recommended number of measurements $\check{E}(X)$ for 9- and 10-server sets. Estimates are produced using CONFIRM for Wisconsin c220g2 servers.

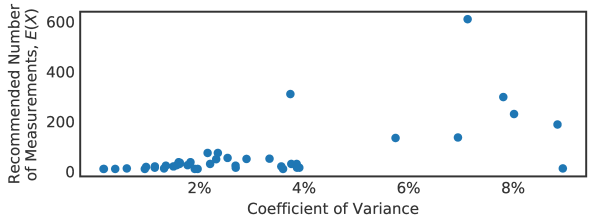


Figure 6: Relationship between CoV and $\check{E}(X)$.

performance measurements. In this and similar cases, not only we are less confident about the value of the statistic of interest—in this case, sample median—but we are likely to make poor conclusions using insufficient number of measurements. Thus, further analysis of the data shown in Table 4 confirms that if we stop after 10 measurements in the 10-server case, our reported median values will be *outside* of the 95% CIs around the medians reported after the recommended 58-68 measurements.

> ♢ ***Use low-variance hardware whenever possible***
> The higher the performance variance of the underlying hardware, the more repetitions must be run to establish statistical significance; conversely, if not enough repetitions are run, there is a greater chance that the conclusions are incorrect.

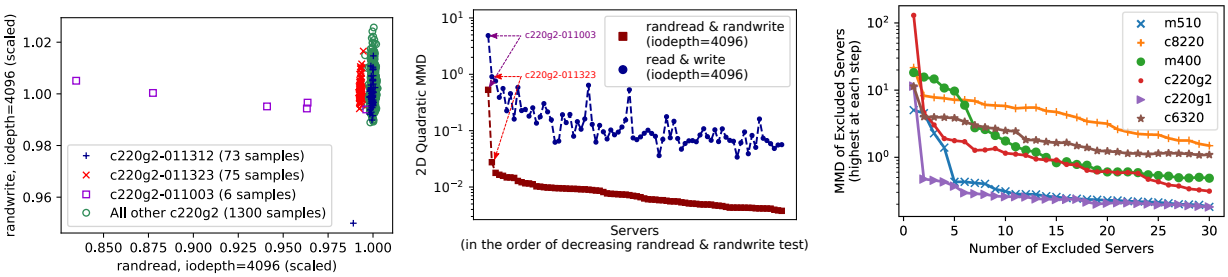
CoV vs. $\check{E}(X)$ Figure 6 shows the relationship between the CoV and the number of repetitions recommended by CONFIRM for the bulk of the configurations from §4.1. This figure is generally favorable for experimenters: most configurations up to about 4% CoV require only tens of repetitions to reach the target of $r = 1\%$ for CIs. Some configurations, however, are extreme outliers, requiring hundreds of experiments to reach this level of confidence. These outliers do not show a consistent pattern in either the type of configuration nor the relationship between CoV and $\check{E}(X)$. The reason that the CoV and $\check{E}(X)$ are not perfectly correlated is that they react differently to outliers and multi-modal distributions. Outliers can skew means and standard deviations quite a bit, but the median is less sensitive to them, and nonparametric CIs effectively take into account the *presence* of points outside the CI but not their *magnitudes*. For extreme multi-modal distributions, such as the one seen in Figure 2, the mean and standard deviation have no problem computing values "in the middle" where no points actually lie, but the median and nonparametric CIs can only pick from points actually in the dataset, making it take much longer for them to converge—or preventing them from converging at all. This figure shows the importance of a tool like CONFIRM: our intuitions about variance, confidence, and the number of repetitions are not always correct, and actual measurements are needed to inform rigorous experiment design.

> ♢ ***Base experiment design on past measurements***
> The relationship between variance and the number of repetitions required is complex; good estimates of the latter require significant prior data.

Using CONFIRM We run CONFIRM as a service at `https://confirm.fyi/` to help users of CloudLab plan their experiments. The tool itself is open-source, so it can be applied to any other facility for which similar data can be collected. We note than when using CONFIRM to estimate the number of repetitions needed for an experiment, it should be used as an *initial* estimate, by selecting the resource(s) that the performance of the experiment is most likely to depend on. Once data is collected, empirical CIs should be computed for the collected data (as described in §2) to ensure that the target allowed error range has been met; the level of variability in a higher-level system may be higher or lower than those found in the low-level benchmarks that CONFIRM uses to compute its estimates.

6 Detecting Unrepresentative Servers

We now turn our attention to the provider's perspective: given what we have seen about the effects of variance on users, what can a provider do to provide resources with consistent performance? As we have seen, some variance is unavoidable, so we pursue the goal of having a *set of servers where every server is representative of the whole*. Put another way, in a distribution drawn from all servers, if we draw samples from a particular server, we should not be able to distinguish those samples from the complete population. This is a strong analysis, as it gets directly at the goal of a testbed or service provider that it should not matter which server(s) an experiment uses: all should provide results that are statistically indistinguishable.



(a) Identifying outlier servers. All values are normalized by the medians.

(b) Ranking servers based on 2D MMD. First two – servers with consistent performance degradation shown in (a).

(c) Using 8 benchmarks (4 disk + 4 memory), we exclude unrepresentative servers for each of the hardware types.

Figure 7: MMD-based server evaluation for c220g2 (a-b) and outlier elimination for all tested hardware types (c).

> ◊ ***Provide indistinguishable resources***
> When servers—even those that are supposedly identical—exhibit performance differences that can be detected reliably through statistical tests, reproducible experimentation is more difficult.

Statistical distributions can be compared based on independent samples using the Mann-Whitney U-test [42]. Unlike its parametric counterpart, the *t*-test, the nonparametric U-test does not assume normality of the compared distributions. As reviewed in [33], many authors have focused on this problem and offered various sophisticated approaches. Appearing in the recent machine learning literature, a kernel[1] two-sample test based on *maximum mean discrepancy* (MMD) [24] offers a powerful solution that is suitable to large-scale datasets and naturally supports multivariate comparisons. This kernel-based testing can be summarized as follows:

The test compares samples $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ from distributions P and Q, where n and m do not need to be equal. No assumption is made about P and Q, and the robustness of this test with different n and m is important for our setting, since we will be using it to compare the samples for an individual server to the rest of the population. MMD provides a measure of similarity (or dissimilarity) between P and Q, expressed as a distance between their embeddings in the reproducing kernel Hilbert space (RKHS) [6]. Abstract in its formulation, this distance metric is still straightforward to use in practice. Similar to many statistical tests, the univariate values obtained using MMD can be compared against thresholds calculated for a given confidence level α and used to estimate probabilities of P and Q being the same distribution given the analyzed samples. The test comes with the quadratic-time and linear-time (w.r.t. $m+n$) estimation variants. The former is a more powerful test as it uses every measurement to the maximum effect, while the latter is more suitable to online processing where the analysis is performed as the data becomes available.

We use the quadratic test implemented in Shogun [57], an open-source machine learning library for Python. One important aspect of MMD testing is kernel selection: we chose a meaningful range of kernel parameters and found that the results of our analysis are not sensitive to particular parameters selected, so we use a common smooth kernel function, a Gaussian kernel,[2] with the bandwidth parameter $\sigma \in [5\%, 50\%]$ of the analyzed measurements. Designed to be robust to individual outliers, MMD tests can point out distribution differences, including pronounced skew and frequent outliers, that are statistically significant.

Based on the MMD statistic, we develop the following method for identifying unrepresentative servers:

Use multiple benchmarks to characterize servers of a particular type. To increase robustness to outliers and avoid bias caused by uneven magnitudes of values in different dimensions, we divide all values by the medians in each dimension prior to kernel testing. Figure 7 (a) demonstrates how such scaled data looks for two disk benchmarks (random read and write tests). In this figure, it is possible to visually identify outlier servers, but it would not be possible to eliminate the outliers cleanly using a simple threshold as the observed distributions overlap (for red and green clusters). In this case, we notice two servers that are unrepresentative—with a small consistent degradation (red) and a larger spread of outlier-like measurements (purple)—in one of the dimensions and a representative server with a single outlier (blue) in the other dimension.

Rank servers: Using the selected benchmarks, we run MMD tests that compare an individual server's samples against samples from all other servers of the same type. This statistic, which represents a measure of dissimilarity,

[1] A *kernel* or a *kernel function* in this context refers to the dot product of features of compared objects.

[2] Gaussian kernel functions facilitate comparison of non-Gaussian distributions and detect differences between multivariate clusters.

is the highest for the least representative servers. In the disk example, the unrepresentative servers end up at the top of the sorted list, as shown in Figure 7 (b). We also observe an expected yet nontrivial result: the same procedure with two different disk benchmarks (sequential tests instead of random), points at performance issues with the same two servers. The exact server ordering in the ranking that uses these sequential tests would be different, but both rankings demonstrate the same elbow-shaped decreasing pattern. At the same time, we confirm that the single-outlier server (blue in Figure 7 (a)) does not show up at the beginning of either ranking as the majority of its samples appear unquestionable.

Eliminate consistent outliers: Actionable insights provided by these dissimilarity rankings allow us to exclude the least representative servers from the pool available to users. We remove them iteratively, one at a time, starting with the least representative server; this ensures that the MMD statistics for the remaining servers are not skewed by the inclusion of the removed servers. Results obtained during such elimination are shown in Figure 7 (c). The elbow-shaped curves indicate that the largest reduction of dissimilarity comes from excluding a few servers at the beginning: from two to seven, representing only 2% of the overall population. Subsequent server elimination provides diminishing returns (note the log scale of the figure).

We have tested this elimination procedure in a variety of settings—in 2D, 4D, and 8D, with each "dimension" being a different configuration—and conclude that the described procedure helps identify the servers with nontrivial performance abnormalities for all analyzed hardware types. The MMD statistic that this test uses is abstract, and does not directly correspond to units in the original space (Gbps, μs, etc.), but this is a necessary side-effect of simultaneously testing metrics that are measured with different units and have different scales; nonetheless, the shape of the curve makes it very clear which servers are not representative. Testbed or service providers can use this procedure to investigate the most unrepresentative servers and take appropriate actions. This method can also help users understand how representative or unrepresentative the servers they use are by revealing their ranks within relevant populations.

7 Steering Clear of Pitfalls

While performing analyses, we ran into situations that resulted in surprising or counter-intuitive results. The potential set of such pitfalls is large, and we have certainly not uncovered all of them, even within the CloudLab environment. However, we can recommend defensive practices that help steer clear of them and likely others.

7.1 ◊ Randomize experiment orderings

Unexpected differences appeared in the memory bandwidth measurements on the two server types at CloudLab Wisconsin: we expected similar results, but the older `c221g1` servers outperformed the newer `c220g2` servers by a factor of nearly 3 (about 36 GB/s versus 12 GB/s) in multi-threaded benchmarks. After a long search, we traced this problem to an unbalanced DIMM configuration in the `c220g2` servers: as a result of their larger memory, the first memory channels were populated with two DIMMs, while the others all had one DIMM. When we had the extra DIMMs removed from one of the `c220g2` servers, memory performance jumped to expected levels. This imbalance appears to interact poorly with a combination of Intel's memory-striping algorithms [28], Linux's allocation of pages in sequential physical order, and the nature of the STREAM benchmark. The result is that STREAM's memory appears to reside mostly or completely on one memory channel, preventing the benchmark from using the hardware's full bandwidth.

While tracking down the cause of this behavior, we found an even more surprising effect: the order in which we ran benchmarks had a dramatic effect on STREAM's performance. In the most extreme case, running a particular benchmark would cause subsequent STREAM runs (until the server was rebooted) to increase their performance by a factor of three, "recovering" approximately the expected performance. Though the exact mechanism behind this recovery is not clear, it appears that the way one benchmark allocates memory—both the size of the allocation and the specific pattern—has an effect on the other's layout on physical channels, so the order in which we run these benchmarks matters. This is an effect that we would not have noticed had we not tried a variety of benchmarks in different orders. Trying to predict ahead of time which orderings would reveal which types of effects would be fruitless; thus a good defensive practice is to **randomize the order of experiments to expose effects that they might have on each other.** Others [48, 45] have made similar observations for other benchmarks.

7.2 ◊ Check configuration sensitivity

The experience related in the previous section also raises another important question: should it be considered a "bug" for a facility like CloudLab to have hardware with an unbalanced memory configuration? Placing blame for the behavior is complex: In the Intel platform, this configuration of DIMMs is legal, but results in fallback to a lower-performance mode that is not widely known. Linux's physical page management policy could also be blamed: FreeBSD does not allocate physical pages sequentially and we found that it exhibits full memory bandwidth performance in this hardware configuration. Our memory benchmarks could also be considered to be at

fault: while they use sufficient RAM to avoid caching issues, they do not use enough to ensure that all DIMMs get exercised. A facility like CloudLab aims to provide servers that are representative of servers in the wider world, and this is a configuration that is not unique to CloudLab.

Ultimately, we believe the primary lesson is the fact that experiments are more sensitive to small details of specific configurations than is commonly acknowledged, and that both facility and user share responsibility for being aware of this sensitivity. The service provider should aim for the highest-quality resources possible. At the same time, it cannot be aware of every interaction between hardware configuration, system software, and workload. The best defensive practice for users is to **perform sensitivity analyses with respect to the hardware configuration**: run experiments on hardware with multiple configurations to understand the extent to which results depend on a particular configuration.

7.3 ♢ Match hardware and software

When we first ran the STREAM benchmarks on the Wisconsin and Clemson servers, we discovered variance that was much higher than we expected. This was because these servers are dual-socket NUMA machines, and STREAM is not NUMA-aware. Not only did this have a deleterious effect on average performance (lowering it 20–25%), but it had an even more pronounced effect on the CoV (raising it from about 80 MB/s to 8,000 MB/s—two orders of magnitude). This problem was simple to resolve: we bind STREAM to one socket at a time, and test each socket separately.

Despite the ease of resolution, this points to a larger problem in experimentation: mismatch between the properties of the hardware and what the software was prepared to handle. Bigger and faster are not always better when it comes to running experiments, and can be worse because they typically imply greater complexity. Experimenters should **carefully consider whether they need features like NUMA, hyperthreading, complex memory hierarchies, etc. before selecting servers that have them**. Using hardware with features not supported in software runs the risk of invalidating results by affecting absolute performance and causing variability that harms the ability to make solid claims backed by statistics.

7.4 ♢ Don't assume independence: check

It is tempting to treat repeated experiments as independent: that earlier experiments do not have an effect on the outcomes of later ones. This is not always the case; one particular instance of this seen in our dataset is the performance of SSDs. Figure 8 shows performance results from a single representative SSD on a `c220g2` server over a period of several months; a clear periodic pattern is present.

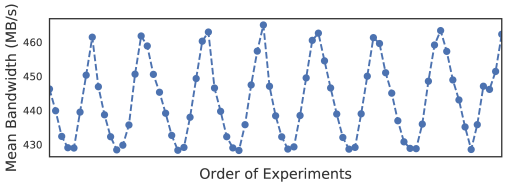


Figure 8: Periodic behavior on a `c220g2` SSD over time for sequential writes with iodepth 4096. Gaps between successive points can represent different durations of time.

Recall that we run `blkdiscard` before every one of these experiments: in theory, this should return the drive to a "clean" state. This periodic behavior seems to be present for two reasons. First, there is likely some sort of "lazy" process that does not do the work of `blkdiscard` all at once but saves part of it for later, resulting in noticeable performance artifacts. Second, this SSD does not seem to be heavily used by other experimenters (it is not the boot disk) so each time we run a new experiment, we are picking up where we left off in the disk's lifecycle.

The effect is that earlier experiments can affect later ones, such as through the quantity of data they write or where they write it, and this effect can persist many weeks later through multiple reboots. Effects may have been even worse if we had not run `blkdiscard`, since this would have left more FTL state from previous experiments. If we assumed independence between runs, we might very well come to incorrect statistical conclusions, as many techniques assume IID (Independent, Identically Distributed) results. This provides more motivation for randomizing the order of experiments, since the sets of experiments that affect one another is not the same for every run. To **test for independence**, we can compare the samples in their original order with with a shuffled version. These comparisons can be done using the Mann-Whitney test or the kernel-based MMD test, similar to the nonparametric two-sample testing we described in §6.

7.5 ♢ Be careful on shared infrastructure

Some experimenters, by choice or necessity, run experiments on virtualized resources in shared environments such as clouds. The most prominent issue with operating in a shared environment is the potential for the presence of "noisy neighbors," whose behavior can impact experimental results, and into which the experimenter has no visibility. Prior work [58, 49, 62, 8] has shown that workloads run by one tenant can affect other tenants in a shared environment. This has implications for variations on three different scales:

- Competing workloads increase variability during their runtime, affecting the *variability seen during individual experiments*.

- Competing workloads may come and go on timescales from minutes to days, causing experiments to get different results *on the same VM at different times*, or changing results during long-running experiments.
- Noisy neighbors may be more prevalent on some hosts than others, making *different VMs* perform differently.

This poses a problem for ensuring accurate experiment results—every bit of additional variance makes it harder to present results with high confidence. In some sense, the presence of a noisy neighbor is analogous to the addition of an "outlier" server as presented in Table 6. To get an intuition for how added variance can affect confidence in results, consider the data reported in Figure 5 (a): this configuration has a CoV of 1.0%, and requires 12 repetitions to achieve the desired confidence. A seemingly modest increase in CoV to 5.0% (Figure 5 (b)) results in a 10× increase in the number of repetitions required (to 121), and a further increase to 8.1% (Figure 5 (c)) requires 670 repetitions (55×). It is also important to note that these calculations assume a *stationary* distribution: that the distribution from which performance results are pulled does not change over time. Clearly, this is not the case with transient noisy neighbors, requiring even more careful experimentation techniques to detect and/or compensate for changing performance characteristics.

Studies have found high CoVs in commercial clouds [18, 29]—particularly for network and disk operations—and the long performance tails in clouds are well-known [14]. Farley et al. [18] found CoVs on EC2 from 0.35% to 25.4% for network bandwidth (average 4.4%), and from 0.5% to 40.9% (average 9.8%) for storage performance. They also found significant differences in performance (typically around 1.2×, but as high as 3.7×) from different VM instances of the same "type" (eg. `m1-small`). Compared with the CoVs found in this study—0.004% CoV for network bandwidth, and average 3.3% CoV (max. 9.86%) for disk I/O—experimenters are likely to require many more repetitions to gain high confidence.

Another issue with running on shared infrastructure is that virtualization adds a layer of abstraction. Even if there are no noisy neighbors to contend with, there is still the presence of the hypervisor. Other studies [49, 62, 56, 11] have explored the extent to which the hypervisor layer impacts the performance of various workloads, including increasing variance.

It is important to note that, as we have explored in this paper, running on non-shared, non-virtualized resources does *not* shield the user entirely from variability: even "bare" hardware has complex, opaque behavior, and the OS kernel can introduce variability just as the hypervisor does. The additional variance from shared resources does not make it *impossible* to run good experiments, but it can make it *much harder*. Earlier work has looked to address issues with running workloads in shared environments. Some solutions [46, 10, 7] focus on the perspective of the provider, and seek to manage these interference effects by varying virtual machine placement or resource allocation. Others [69] approach this from the perspective of the client and try to find the "best" type of virtual machine. The common thread between these solutions, however, is the reality that performance interference effects must be *managed* and cannot be entirely *avoided*. To achieve statistical confidence, the experimenter is likely to have to run many more experiments, and to consider sources of variation that are not stationary, which makes experiment design far more complex. Conversely, experiments run in this environment that do *not* account for increased and more complex variance run a larger risk of coming to incorrect conclusions: for a fixed number of runs, the more variance is present, the wider the confidence intervals. The wider the confidence intervals, the larger the effect that can be potentially misreported.

Our overall recommendation is to **run experiments in a shared (and therefore, likely high-variance) environment only if it is unavoidable.** If experiments must be run in such an environment, design them in ways that help compensate for variability: run many more repetitions, run on many different VMs and at different times to avoid over-measuring artifacts from particular neighbors, and ensure that the experiment design does not introduce systematic bias.

7.6 ♢ Plan experiments for uncertainty

It is not always practical to run a large number of repetitions of an experiment. This can be due to factors such as monetary costs, long execution times or both. Techniques in Active Learning [53] and Bayesian Optimization [54] help design sequences of experiments that efficiently "explore" available configurations. Generally speaking, the former class of techniques focuses on reducing the uncertainty about experiments' outcomes, while the latter helps find configurations corresponding to the maximums (or minimums) of the objective functions studied via experimentation. In contrast with classical (static) experiment design, these iterative techniques train Machine Learning models on the data available from existing experiments and use the recommendations produced by these models to run subsequent additional experiments. There is a wealth of literature describing optimizations for these techniques, including [20] and [36], as well as specific computer applications, such as [16], [22], and [4], among many other studies. While these experimentation techniques are mostly outside the scope of this study, as part of our future work, we intend to equip CONFIRM with the ability to recommend specific servers and specific

hardware and benchmark configurations for additional experiments on the basis of high performance variability and observed outliers.

8 Related Work

In [37], the authors present a profiling study of a Warehouse-Scale Computer where they analyze 12 to 36 months worth of performance counter metrics for applications running on Google data centers. The study focuses on microarchitecture-level statistics to identify hotspots in distributed applications, main memory and CPU cache latencies, among others. In contrast, we focus on coarser-grained metrics such as runtime and bandwidth of microbenchmarks with the goal of taking into account the points of view of both system administrators and users. Similar studies have focused on other cloud platforms such as Microsoft's Azure [39]. Other related profiling efforts have the goal of improving the scheduling of applications on shared infrastructure by identifying and reducing contention between applications [35, 70]. More recently, in [25], the authors present a study of the impact of slow failures (i.e. "hardware that is still running and functional but in a degraded mode, slower than its expected performance") found in large-scale cluster deployments in 12 institutions.

In [48] the authors describe a suite of tests composed of of microbenchmarks that run continuously over the entire Grid5000 infrastructure. The heuristic to decide which tests to run and where is similar to ours, but in our case we prioritize testbed coverage. In [47] a set of open questions for experimental testbeds are outlined, with respect to reproducibility of experiments. In particular, the topic of "Respective Responsibilities of Testbeds and Experimenters" poses the questions of "How far should testbeds go with providing advanced services to experimenters? What should be left as a burden for experimenters?" As part of our work, we have introduced the foundation for a new service that aids experimenters in getting a better understanding of the variability of the underlying platform with respect to the performance of basic subcomponents (CPU, memory bandwidth, network and storage).

Another two broad topics that relate to our work are anomaly detection [9, 64, 63] and straggler analysis [14, 2, 68]. In the former, runtime metrics are analyzed either offline or online in order to identify events that do not conform with the performance expectation of the operator, either at hardware or software levels. Straggler analysis deals with identifying a small proportion of subjobs that cause significant degradations on the parent job. We see our work as complementary to these two topics and envision the methodology and analysis presented here as a way of generating a baseline on which new techniques and approaches in both can be evaluated.

DCBench [32], CloudSuite [19], TailBench [38] and BigDataBench [66] are benchmarking suites whose goal is to recreate workloads that run on cloud infrastructures. In our case, our goal was to target any type of workload running on CloudLab and thus we ended up selecting a generic (and simple) workload for our study.

9 Conclusion and Future Work

In this paper, we have explored the types and magnitudes of hardware performance variation that are an inevitable part of measuring the performance of computer systems. The method we developed for finding unrepresentative resources can be used to provide more consistent environments, and the CONFIRM system can help to design better experiments. These results demonstrate valuable properties of a large, shared experimentation platform: scale is required in order to determine which servers are representative and which are not, and measurement and analysis done once can be used for many experiments.

In this study, we have deliberately focused on the set of hardware resources whose performance is of the most interest in the CloudLab testbed. Differences due to system software and libraries—kernels, compilers, memory allocators, etc. should not be discounted, and there are many more hardware metrics that are of interest. We hope to expand our study to include these factors in the future.

Code and Data

Raw Data and Analysis Code:
`doi:10.5281/zenodo.1435969`
CONFIRM: `https://gitlab.flux.utah.edu/emulab/confirm`
Benchmarks: `https://gitlab.flux.utah.edu/emulab/cloudlab-benchmarks`
Benchmark Orchestration: `https://gitlab.flux.utah.edu/emulab/cloudlab-orchestration`

Specific versions within the `git` repositories used for this paper are identified with the `osdi18` tag.

Acknowledgments

We would like to thank Jeff Phillips for suggesting the Kernel 2-sample MMD test for our evaluation, as well as providing valuable assistance in understanding it. We are grateful to the faculty and staff of the Flux Research Group for their feedback prior to submission, and to the anonymous OSDI reviewers as well as our shepherd, Justine Sherry, for their feedback and suggestions during the review and shepherding process. This work was made possible by the CloudLab testbed, supported by the National Science Foundation under Grant Nos. CNS-1419199 and CNS-1743363. This work was also partially supported by NSF Grant No. OAC-1450488 and the Center for Research in Open Source Software (`https://cross.ucsc.edu`).

References

[1] J. Allspaw. *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, Inc., 2008.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[3] J. Axboe. Flexible I/O tester. `https://github.com/axboe/fio`, 2006-2018.

[4] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[5] Barnstormer Softworks, Ltd. Welcome to geni-lib's documentation! `http://docs.cloudlab.us/geni-lib/index.html`, 2016.

[6] A. Berlinet and C. Thomas-Agnan. *Reproducing kernel Hilbert spaces in probability and statistics*. Springer Science & Business Media, 2011.

[7] F. Caglar, S. Shekhar, and A. S. Gokhale. Towards a performance interference-aware virtual machine placement strategy for supporting soft real-time applications in the cloud. In *REACTION*, 2014.

[8] G. Casale, S. Kraft, and D. Krishnamurthy. A model of storage I/O performance interference in virtualized systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, pages 34–39. IEEE Computer Society, 2011.

[9] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[10] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 164–173, Oct 2015.

[11] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, ATC '05, pages 24–24. USENIX Association, 2005.

[12] C. Curtsinger and E. D. Berger. Stabilizer: statistically sound performance evaluation. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 219–228. ACM, 2013.

[13] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Why you should care about quantile regression. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[14] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[15] D. A. Dickey and W. A. Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366a):427–431, 1979.

[16] D. Duplyakin, J. Brown, and D. Calhoun. Evaluating active learning with cost and memory awareness. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 214–223, May 2018.

[17] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: why some (might) like it hot. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):163–174, 2012.

[18] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, Oct. 2012.

[19] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[20] W. Fu, M. Wang, S. Hao, and X. Wu. Scalable active learning by approximated error reduction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1396–1405. ACM, 2018.

[21] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[22] R. B. Gramacy and H. K. Lee. Adaptive design and analysis of supercomputer experiments. *Technometrics*, 51(2):130–145, 2009.

[23] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013.

[24] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[25] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, 2018.

[26] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The unwritten contract of solid state drives.

In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144. ACM, 2017.

[27] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 73. ACM, 2015.

[28] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 248966-040 edition, April 2018. Section 2.4.6.

[29] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *Proceedings of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 104–113, 2011.

[30] iPerf3 Authors. iPerf - the ultimate speed test tool for TCP, UDP and SCTP. `https://iperf.fr/`.

[31] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, Apr. 1991.

[32] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo. Characterizing data analysis workloads in data centers. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 66–76. IEEE, 2013.

[33] J. Jurečková, J. Kalina, et al. Nonparametric multivariate rank tests and their unbiasedness. *Bernoulli*, 18(1):229–251, 2012.

[34] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 484–490, 2005.

[35] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[36] K. Kandasamy, J. Schneider, and B. Póczos. Bayesian active learning for posterior estimation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[37] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 158–169. ACM, 2015.

[38] H. Kasture and D. Sanchez. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[39] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE micro*, (4):8–19, 2010.

[40] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[41] J.-Y. Le Boudec. *Performance evaluation of computer and communication systems*. EPFL Press, 2011.

[42] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pages 50–60, 1947.

[43] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[44] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 177–190. ACM, 2015.

[45] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[46] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250. ACM, 2010.

[47] L. Nussbaum. Testbeds support for reproducible research. In *Proceedings of the Reproducibility Workshop*, pages 24–26. ACM, 2017.

[48] L. Nussbaum. Towards trustworthy testbeds thanks to throughout testing. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pages 1571–1578, 2017.

[49] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao. Who is your neighbor: Net I/O performance interference in virtualized clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, July 2013.

[50] A. W. Reece. Achieving maximum memory bandwidth. `http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html`, May 18 2013.

[51] A. W. Reece. Memory bandwidth demo. `https://github.com/awreece/memory-bandwidth-demo`, May 19 2013.

[52] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 193–204. ACM, 2009.

[53] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison, 2009.

[54] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[55] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometricka*, 52:591–611, Dec. 1965.

[56] R. Shea, F. Wang, H. Wang, and J. Liu. A deep investigation into network performance in virtual machine based cloud environments. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1285–1293, April 2014.

[57] S. Sonnenburg, H. Strathmann, S. Lisitsyn, V. Gal, F. J. I. Garcìa, W. Lin, S. De, C. Zhang, frx, tklein23, E. Andreev, JonasBehr, sploving, P. Mazumdar, C. Widmer, P. D. . Zora, G. D. Toni, S. Mahindre, A. Kislay, K. Hughes, R. Votyakov, khalednasr, S. Sharma, A. Novik, A. Panda, E. Anagnostopoulos, L. Pang, A. Binder, serialhex, and B. Esser. Shogun 6.1.0, Nov. 2017.

[58] J. Taheri, A. Y. Zomaya, and A. Kassler. vmbbprofiler: A black-box profiling approach to quantify sensitivity of virtual machines to shared cloud resources. *Computing*, 99(12):1149–1177, Dec. 2017.

[59] The CloudLab Team. CloudLab hardware. `https://www.cloudlab.us/hardware.php`, 2018.

[60] The CloudLab Team. The cloudlab testbed. `https://cloudlab.us/`, 2018.

[61] The CloudLab Team. Hardware. `http://docs.cloudlab.us/hardware.html`, 2018.

[62] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, Jan. 2010.

[63] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *Network Operations and Management Symposium (NOMS)*, pages 96–103. IEEE, 2010.

[64] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. Statistical techniques for online anomaly detection in data centers. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 385–392. IEEE, 2011.

[65] G. Wang, L. Zhang, and W. Xu. What can we learn from four years of data center hardware failures? In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 25–36. IEEE, 2017.

[66] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[67] N. J. Wright, S. Smallen, C. M. Olschanowsky, J. Hayes, and A. Snavely. Measuring and understanding variation in benchmark performance. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 438–443. IEEE, 2009.

[68] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*. ACM, 2014.

[69] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 452–465. ACM, 2017.

[70] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

Pocket: Elastic Ephemeral Storage for Serverless Analytics

Ana Klimovic[1] Yawen Wang[1] Patrick Stuedi [2]

Animesh Trivedi[2] Jonas Pfefferle[2] Christos Kozyrakis[1]

[1] Stanford University [2] IBM Research

Abstract

Serverless computing is becoming increasingly popular, enabling users to quickly launch thousands of short-lived tasks in the cloud with high elasticity and fine-grain billing. These properties make serverless computing appealing for interactive data analytics. However exchanging intermediate data between execution stages in an analytics job is a key challenge as direct communication between serverless tasks is difficult. The natural approach is to store such ephemeral data in a remote data store. However, existing storage systems are not designed to meet the demands of serverless applications in terms of elasticity, performance, and cost. We present *Pocket*, an elastic, distributed data store that automatically scales to provide applications with desired performance at low cost. Pocket dynamically rightsizes resources across multiple dimensions (CPU cores, network bandwidth, storage capacity) and leverages multiple storage technologies to minimize cost while ensuring applications are not bottlenecked on I/O. We show that Pocket achieves similar performance to ElastiCache Redis for serverless analytics applications while reducing cost by almost 60%.

1 Introduction

Serverless computing is becoming an increasingly popular cloud service due to its high elasticity and fine-grain billing. Serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions enable users to quickly launch thousands of light-weight tasks, as opposed to entire virtual machines. The number of serverless tasks scales automatically based on application demands and users are charged only for the resources their tasks consume, at millisecond granularity [17, 36, 56].

While serverless platforms were originally developed for web microservices and IoT applications, their elasticity and billing advantages make them appealing for data intensive applications such as *interactive analytics*. Several recent frameworks launch large numbers of fine-grain tasks on serverless platforms to exploit all available parallelism in an analytics job and achieve near real-time performance [32, 45, 27]. In contrast to traditional serverless applications that consist of a single function executed when a new request arrives, analytics jobs typically consist of multiple stages and require sharing of state and data across stages of tasks (e.g., data shuffling).

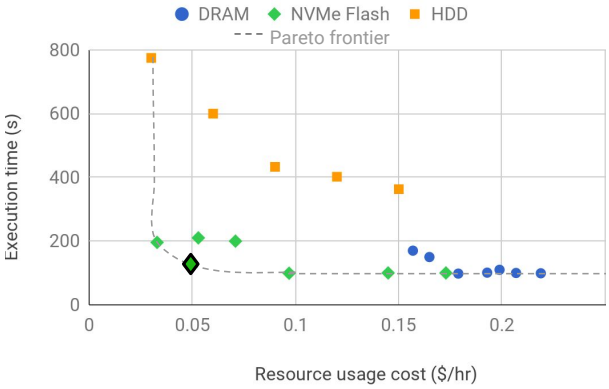


Figure 1: Example of performance-cost trade-off for a serverless video analytics job using different storage technologies and VM types in Amazon EC2

Most analytics frameworks (e.g., Spark) implement data sharing with a long-running framework agent on each node buffering intermediate data in local storage [78]. This enables tasks from different execution stages to directly exchange intermediate data over the network. However, in serverless deployments, there is no long-running application framework agent to manage local storage. Furthermore, serverless applications have no control over task scheduling or placement, making direct communication among tasks difficult. As a result of these limitations, the natural approach for data sharing in serverless applications is to use a remote storage service. For instance, early frameworks for serverless analytics either use object stores (e.g., S3 [16]), databases (e.g., CouchDB [1]) or distributed caches (e.g., Redis [51]).

Unfortunately, existing storage services are not a good fit for sharing short-lived intermediate data in serverless applications. We refer to the intermediate data as

ephemeral data to distinguish it from input and output data which requires long-term storage. File systems, object stores and NoSQL databases prioritize providing durable, long-term, and highly-available storage rather than optimizing for performance and cost. Distributed key-value stores offer good performance, but burden users with managing the storage cluster scale and configuration, which includes selecting the appropriate compute, storage and network resources to provision.

The availability of different storage technologies (e.g., DRAM, NVM, Flash, and HDD) increases the complexity of finding the best cluster configuration for performance and cost. However, the choice of storage technology is critical since jobs may exhibit different storage latency, bandwidth and capacity requirements while different storage technologies vary significantly in terms of their performance characteristics and cost [48]. As an example, Figure 1 plots the performance-cost tradeoff for a serverless video analytics application using a distributed ephemeral data store configured with different storage technologies, number of nodes, compute resources per node, and network bandwidth (see §6.1 for our AWS experiment setup). Each resource configuration leads to different performance and cost. Finding Pareto efficient storage allocations for a job is non-trivial and gets more complicated with multiple jobs.

We present *Pocket*, a distributed data store designed for efficient data sharing in serverless analytics. Pocket offers high throughput and low latency for arbitrary size data sets, automatic resource scaling, and intelligent data placement across multiple storage tiers such as DRAM, Flash, and disk. The unique properties of Pocket result from a strict separation of responsibilities across three planes: a control plane which determines data placement policies for jobs, a metadata plane which manages distributed data placement, and a 'dumb' (i.e., metadata-oblivious) data plane responsible for storing data. Pocket scales all three planes independently at fine resource and time granularity based on the current load. Pocket uses heuristics, which take into account job characteristics, to allocate the right storage media, capacity, bandwidth and CPU resources for cost and performance efficiency. The storage API exposes deliberately simple I/O operations for sub-millisecond access latency. We intend for Pocket to be managed by cloud providers and offered to users with a pay-what-you-use cost model.

We deploy Pocket on Amazon EC2 and evaluate the system using using three serverless analytics workloads: video analytics, MapReduce sort, and distributed source code compilation. We show that Pocket is capable of rightsizing the type and number of resources such that jobs achieve similar performance compared to using ElastiCache Redis, a DRAM-based key-value store, while saving almost 60% in cost.

In summary, our contributions are as follows:

- We identify the key characteristics of ephemeral data in serverless analytics and synthesize requirements for storage platforms used to share such data among serverless tasks.
- We introduce Pocket, a distributed data store whose control, metadata and data planes are designed for sub-second response times, automatic resource scaling and intelligent data placement across storage tiers. To our knowledge, Pocket is the first platform targeting data sharing in serverless analytics.
- We show that Pocket's data plane delivers sub-millisecond latency and scalable bandwidth while the control plane rightsizes resources based on the number of jobs and their attributes. For a video analytics job, Pocket reduces the average time serverless tasks spend on ephemeral I/O by up to 4.1× compared to S3 and achieves similar performance to ElastiCache Redis while saving 59% in cost.

Pocket is open-source software. The code is available at: `https://github.com/stanford-mast/pocket`.

2 Storage for Serverless Analytics

Early work in serverless analytics has identified the challenge of storing and exchanging data between hundreds of fine-grain, short-lived tasks [45, 32]. We build on our study of ephemeral storage requirements for serverless analytics applications [49] to synthesize essential properties for an ephemeral data storage solution. We also discuss why current systems are not able to meet the ephemeral I/O demands of serverless analytics applications. Our focus is on ephemeral data as the original input and final output data of analytics jobs typically has long-term availability and durability requirements that are well served by the variety of file systems, object stores, and databases available in the cloud.

2.1 Ephemeral Storage Requirements

High performance for a wide range of object sizes: Serverless analytics applications vary considerably in the way they store, distribute, and process data. This diversity is reflected in the granularity of ephemeral data that is generated during a job. Figure 2 shows the ephemeral object size distribution for a distributed lambda compilation of the `cmake` program, a serverless video analytics job using the Thousand Island (THIS) video scanner [63], and a 100 GB MapReduce sort job on lambdas. The key observation is that ephemeral data access granularity varies greatly in size, ranging from hundreds of bytes to hundreds of megabytes. We observe a

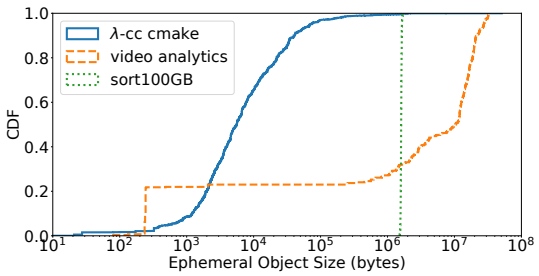


Figure 2: Objects are 100s of bytes to 100s of MBs.

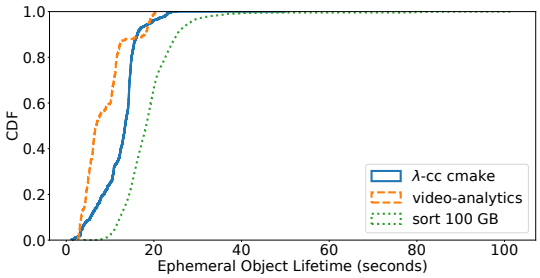


Figure 3: Objects have short lifetime.

straight line for sorting as its ephemeral data size is equal to the partition size. However, with a different dataset size and/or number of workers, the location of the line changes. Applications that read/write large objects demand high throughput (e.g., we find that sorting 100 GB with 500 lambdas requires up to 7.5 GB/s of ephemeral storage throughput) while low latency is important for small object accesses. *Thus, an ephemeral data store must deliver high bandwidth, low latency, and high IOPS for the entire range of object sizes.*

Automatic and fine-grain scaling: One of the key promises of serverless computing is agility to dynamically meet application demands. Serverless frameworks can launch thousands of short-lived tasks instantaneously. Thus, an ephemeral data store for serverless applications can observe a storm of I/O requests within a fraction of a second. Once the load dissipates, the storage (just like the compute) resources should be scaled down for cost efficiency. Scaling up or down to meet elastic application demands requires a storage solution capable of growing and shrinking in multiple resource dimensions (e.g., adding/removing storage capacity and bandwidth, network bandwidth, and CPU cores) at a fine time granularity on the order of seconds. In addition, users of serverless platforms desire a storage service that automatically manages resources and charges users only for the fine-grain resources their jobs actually consume, so as to match the abstraction that serverless computing already provides for compute and memory resources. Automatic resource management is important since navigating cluster configuration performance-cost trade-offs is a burden for users. For example, finding the Pareto optimal point outlined in Figure 1 is non-trivial; it is the point beyond which adding resources only increases cost without improving execution time while using any lower-cost resource allocation results in sub-optimal execution time. In summary, *an ephemeral data store must automatically rightsize resources to satisfy application I/O requirements while minimizing cost.*

Storage technology awareness: In addition to rightsizing cluster resources, the storage system also needs to decide which storage technology to use for which data. The variety of storage media available in the cloud allow for different performance-cost trade-offs, as shown in Figure 1. Each storage technology differs in terms of I/O latency, throughput and IOPS per GB of capacity, and the cost per GB. The optimal choice of storage media for a job depends on its characteristics. *Hence, the ephemeral data store must place application data on the right storage technology tier(s) for performance and cost efficiency.*

Fault-(in)tolerance: Typically a data store must deal with failures while keeping the service up and running. Hence, it is common for storage systems to use fault-tolerance techniques such as replication and erasure codes [42, 66, 46]. For data that needs to be stored long-term, such as the original input and final output data for analytics workloads, the cost of data unavailability typically outweighs the cost of fault-tolerance mechanisms. However, as shown in Figure 3, ephemeral data has a short lifetime of 10-100s of seconds. Unlike the original input and final output data, ephemeral data is only valuable during the execution of a job and can easily be regenerated. Furthermore, fault tolerance is typically baked into compute frameworks, such that storing the data and computing it become interchangeable [39]. For example, Spark uses a data abstraction called resilient distributed datasets (RDDs) to mitigate stragglers and track lineage information for fast data recovery [78]. *Hence, we argue that an ephemeral storage solution does not have to provide high fault-tolerance as expected of traditional storage systems.*

2.2 Existing Systems

Existing storage systems do not satisfy the combination of requirements outlined in § 2.1. We describe different categories of systems and summarize why they fall short for elastic ephemeral storage in Table 1.

Severless applications commonly use fully-managed cloud storage services, such as Amazon S3, Google Cloud Storage, and DynamoDB. These systems extend the 'serverless' abstraction to storage, charging users only for the capacity and bandwidth they use [16, 28].

	Elastic scaling	Latency	Throughput	Max object size	Cost
S3	Auto, coarse-grain	High	Medium	5 TB	$
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	$$
Elasticache Redis	Manual	Low	High	512 MB	$$$
Aerospike	Manual	Low	High	1 MB	$$
Apache Crail	Manual	Low	High	any size	$$
Desired for λs	*Auto, fine-grain, pay per second*	*Low*	*High*	*any size*	*$*

Table 1: Comparison of existing storage systems and desired properties for ephemeral storage in serverless analytics.

While such services automatically scale resources based on usage, they are optimized for high durability hence their agility is limited and they do not meet the performance requirements of serverless analytics applications. For example, S3 has high latency overhead (e.g., a 1 KB read takes ∼12 ms) and insufficient throughput for highly parallel applications. For example, sorting 100 GB with 500 or more workers results in request rate limit errors when S3 is used for intermediate data.

In-memory key-value stores, such as Redis and Memcached, provide another option for storing ephemeral data [51, 8]. These systems offer low latency and high throughput but at the higher cost of DRAM. They also require users to manage their own storage instances and manually scale resources. Although Amazon and Azure offer managed Redis clusters through their ElastiCache and Redis Cache services respectively, they do not automate storage management as desired by serverless applications [13, 57]. Users must still select instance types with the appropriate memory, compute and network resources to match their application requirements. In addition, changing instance types or adding/removing nodes can require tearing down and restarting clusters, with nodes taking minutes to start up while the service is billed for hourly usage.

Another category of systems use Flash storage to decrease cost, while still offering good performance. For example, Aerospike is a popular Flash-based NoSQL database [69]. Alluxio/Tachyon is designed to enable fast and fault-tolerant data sharing between multiple jobs [53]. Apache Crail is a distributed storage system that uses multiple media tiers to balance performance and cost [2]. Unfortunately, users must manually configure and scale their storage cluster resources to adapt to elastic job I/O requirements. Finding Pareto optimal deployments for performance and cost efficiency is non-trivial, as illustrated for a single job in Figure 1. Cluster configuration becomes even more complex when taking into account the requirements of multiple overlapping jobs.

3 Pocket Design

We introduce *Pocket*, an elastic distributed storage service for ephemeral data that automatically and dynamically rightsizes storage cluster resource allocations to provide high I/O performance while minimizing cost. Pocket addresses the requirements outlined in §2.1 by applying the following key design principles:

1. **Separation of responsibilities:** Pocket divides responsibilities across three different planes: the control plane, the metadata plane, and the data plane. The control plane manages cluster sizing and data placement. The metadata plane tracks the data stored across nodes in the data plane. The three planes can be scaled independently based on variations in load, as described in §4.2.

2. **Sub-second response time:** All I/O operations are deliberately simple, targeting sub-millisecond latencies. Pocket's storage servers are optimized for fast I/O and are only responsible for storing data (not metadata), making them simple to scale up or down. The controller scales resources at second granularity and balances load by intelligently steering incoming job data. This makes Pocket elastic.

3. **Multi-tier storage:** Pocket leverages different storage media (DRAM, Flash, disk) to store a job's data in the tier(s) that satisfy the I/O demands of the application while minimizing cost (see §4.1).

3.1 System Architecture

Figure 4 shows Pocket's system architecture. The system consists of a logically centralized controller, one or more metadata servers, and multiple data plane storage servers.

The controller, which we describe in §4, allocates storage resources for jobs and dynamically scales Pocket metadata and storage nodes up and down as the number

of jobs and their requirements vary over time. The controller also makes data placement decisions for jobs (i.e., which nodes and storage media to use for a job's data).

Metadata servers enforce coarse-grain data placement policies generated by the controller by steering client requests to appropriate storage servers. Pocket's metadata plane manages data at the granularity of *blocks*, whose size is configurable. We use a 64 KB block size in our deployment. Objects larger than the block size are divided into blocks and distributed across storage servers, enabling Pocket to support arbitrary object sizes. Clients access data blocks on metadata-oblivious, performance-optimized storage servers equipped with different storage media (DRAM, Flash, and/or HDD).

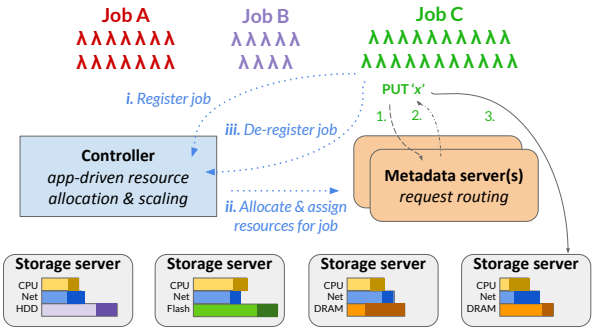


Figure 4: Pocket system architecture and the steps to register job C, issue a PUT from a lambda and de-register the job. The colored bars on storage servers show used and allocated resources for all jobs in the cluster.

3.2 Application Interface

Table 2 outlines Pocket's application interface. Pocket exposes an object store API with additional functions tailored to the ephemeral storage use-case. We describe these functions and how they map to Pocket's separate control, metadata and data planes.

Control functions: Applications use two API calls, `register_job` and `deregister_job`, to interact with the Pocket controller. The `register_job` call accepts hints about a job's characteristics (e.g., degree of parallelism, latency-sensitivity) and requirements (e.g., capacity, throughput). These optional hints help the controller rightsize resource allocations to optimize performance and cost (see §4.1). The `register_job` call returns a job identifier and the metadata server(s) assigned for managing the job's data. The `deregister_job` call notifies the controller that a serverless job has completed.

Metadata functions: While control API calls are issued once per job, serverless tasks in a job can interact with Pocket metadata servers multiple times during their lifetime to write and read ephemeral data. Serverless clients use the `connect` call to establish a connection with Pocket's metadata service. Data in Pocket is stored in objects which are organized in buckets. Objects and buckets are identified using names (strings). Clients can create and delete buckets and enumerate objects in buckets by passing their job identifier and the bucket name. Clients can also lookup and delete existing objects. These metadata operations are similar to those supported by other object stores like Amazon S3.

In our current design, Pocket stores all of a job's data in a top-level bucket identified by the job's ID, which is created during job registration by the controller. This implies each job is assigned to a single metadata server, since a bucket is only managed by one metadata server, to simplify consistency management. However, a job is not fundamentally limited to one metadata server. In general, jobs can create multiple top-level buckets which hash to different metadata servers. In §6.2 we show that a single metadata server in our deployment supports 175K requests per second, which for the applications we study is sufficient to support jobs with thousands of lambdas.

Storage functions: Clients `put` and `get` data to/from objects at a byte granularity. Clients provide their job identifier for all operations. `Put` and `get` operations first involve a metadata lookup. Pocket enhances the basic `put` and `get` object store API calls by accepting an optional data lifetime management hint for these two calls. Since ephemeral data is usually only valuable during the execution of a job, Pockets default coarse-grained behavior is to delete a job's data when the job deregisters. However, applications can set flags to override the default deletion policy for particular objects.

If a client issues a `put` with the PERSIST flag set to true, the object will persist after the job completes. The object is stored on long-running Pocket storage nodes (see §4.2) and will remain in Pocket until it is explicitly deleted or a (configurable) timeout period has elapsed. The ability to persist objects beyond the duration of a job is useful for piping data between jobs. If a client issues a `get` with the DELETE flag set to true, the object will be deleted as soon as it is read, allowing for more efficient garbage collection. Our analysis of ephemeral I/O characteristics for serverless analytics applications reveals that ephemeral data is often written and read only once. For example, a mapper writes an intermediate object destined to a particular reducer. Such data can be deleted as soon as it is consumed instead of waiting for the job to complete and deregister.

3.3 Life of a Pocket Application

We now walk through the life of a serverless analytics application using Pocket. Before launching lambdas, the application first registers with the controller and option-

Client API Function	Description
register_job(jobname, hints=None)	register job with controller and provide optional hints, returns a job ID and metadata server IP address
deregister_job(jobid)	notify controller job has finished, delete job's non-PERSIST data
connect(metadata_server_address)	open connection to metadata server
close()	close connection to metadata server
create_bucket(jobid, bucketname)	create a bucket
delete_bucket(jobid, bucketname)	delete a bucket
enumerate(jobid, bucketname)	enumerate objects in a bucket
lookup(jobid, obj_name)	return `true` if obj_name data exists, else `false`
delete(jobid, obj_name)	delete data
put(jobid, src_filename, obj_name, PERSIST=`false`)	write data, set PERSIST flag if want data to remain after job finishes
get(jobid, dst_filename, obj_name, DELETE=`false`)	read data, set DELETE `true` if data is only read once

Table 2: Main control, metadata, and storage functions exposed by Pocket's client API.

ally provides hints about the job's characteristics (step i in Figure 4). The controller determines the storage tier to use (DRAM, Flash, disk) and the number of storage servers across which to distribute the job's data to meet its throughput and capacity requirements. The controller generates a weight map, described in §4.1, to specify the job's data placement policy and sends this information to the metadata server which it assigned for managing the job's metadata and steering client I/O requests (step ii). If the controller needs to launch new storage servers to satisfy a job's resource allocation, the job registration call stalls until these nodes are available.

When registration is complete, the job launches lambdas. Lambdas first connect to their assigned metadata server, whose IP address is provided by the controller upon job registration. Lambda clients write data by first contacting the metadata server to get the IP address and block address of the storage server to write data to. For writes to large objects which span multiple blocks, the client requests capacity allocation from the metadata server in a streaming fashion; when the capacity of a single block is exhausted, the client issues a new capacity allocation request to the metadata server. Pocket's client library internally overlaps metadata RPCs for the next block while writing data for the current block to avoid stalls. Similarly, lambdas read data by first contacting the metadata server in a similar fashion. Clients cache metadata in case they need to read an object multiple times.

When the last lambda in a job finishes, the job deregisters the job to free up Pocket resources (step iii). Meanwhile, as jobs execute, the controller ontinuously monitors resource utilization in storage and metadata servers (the horizontal bars on storage servers in Figure 4) to add/remove servers as needed to minimize cost while providing high performance (see §4.2).

3.4 Handling Node Failures

Though Pocket is not designed to provide high data durability, the system has mechanisms in place to deal with node failures. Storage servers send heartbeats to the controller and metadata servers. When a storage server fails to send heartbeats, metadata servers automatically mark its blocks as invalid. As a result, client read operations to data that was stored on the faulty storage server will return a 'data unavailable' error. Pocket currently expects the application framework to re-launch serverless tasks to regenerate lost ephemeral data. A common approach is for application frameworks to track data lineage, which is the sequence of tasks that produces each object [78, 39]. For metadata fault tolerance, Pocket supports logging of all metadata RPC operations on shared storage. When a metadata server fails, its state can be reconstructed by replaying the shared log. Controller fault tolerance can be achieved through master-slave replication, though we do not evaluate this in our study.

4 Rightsizing Resource Allocations

Pocket's control plane elastically and automatically rightsizes cluster resources. When a job registers, Pocket's controller leverages optional hints passed through the API to conservatively estimate the job's latency, throughput and capacity requirements and find a cost-effective resource assignment, as described in §4.1. In addition to rightsizing resource allocations for jobs upfront, Pocket continuously monitors the cluster's overall utilization and decides when and how to scale storage and metadata nodes based on load. §4.2 describes Pocket's resource scaling mechanisms along with its data steering policy to balance load.

Hint	Impact on throughput T	Impact on capacity C	Impact on storage media
No hint (default policy)	$T = T_{\text{default}}$ ($T = 50\times$ 8 Gb/s)	$C = C_{\text{default}}$ ($C = 50\times$ 1960 GB)	Fill storage tiers in order of high to low performance (DRAM first, then Flash)
Latency sensitivity	-	-	If latency sensitive, use default policy above. Otherwise, choose the storage tier with the lowest cost for the estimated throughput T and capacity C required for the job.
Maximum number of concurrent lambdas N	$T = N\times$ per-λ Gb/s limit ($T = N \times 0.6$ Gb/s)	$C \propto N\times$ per-λ Gb/s limit ($C = \frac{N\times 0.6}{8\text{ Gb/s}} \times$ 1960 GB)	
Total ephemeral data capacity D	$T \propto D$ ($T = \frac{D}{1960\text{GB}} \times 8$ Gb/s)	$C = D$	
Peak aggregate bandwidth B	$T = B$	$C \propto B$ ($C = \frac{B}{8\text{ Gb/s}} \times 1960$GB)	

Table 3: The impact that hints provided about the application have on Pocket's resource allocation decisions for throughput, capacity and the choice of storage media (with specific examples in parentheses for our AWS deployment with `i3.2xl` instances, each with 8 cores, 60 GB DRAM, 1.9 TB Flash and ∼8 Gb/s network bandwidth).

4.1 Rightsizing Application Allocation

When a job registers, the controller first determines its *resource allocation* across three dimensions: throughput, capacity, and the choice of storage media. The controller then uses an online bin-packing algorithm to translate the resource allocation into a *resource assignment* on nodes.

Determining job I/O requirements: Pocket uses heuristics that adapt to optional hints passed through the `register_job` API. Table 3 lists the hints that Pocket supports and their impact on the throughput, capacity, and choice of storage media allocated for a job, with examples (in parentheses) for our deployment on AWS.

Given no hints about a job, Pocket uses a default resource allocation that conservatively over-provisions resources to achieve high performance, at high cost. In our AWS deployment, this consists of 50 `i3.2xl` nodes, providing DRAM and NVMe Flash storage with 50 GB/s aggregate throughput. By default, Pocket conservatively assumes that a job is latency sensitive. Hence, Pocket fills the job's DRAM resources before spilling to other storage tiers, in order of increasing storage latency. If a job hints that it is not sensitive to latency, the controller does not allocate DRAM for the job and instead uses the most cost-effective storage technology for the throughput and capacity the controller estimates the job needs.

Knowing a job's maximum number of concurrent lambdas, N, allows Pocket to compute a less conservative estimate of the job's throughput requirement. If this hint is provided, Pocket allocates throughput equal to N times the peak network bandwidth limit per lambda (e.g., ∼600 Mb/s per lambda on AWS). N can be limited by the job's inherent parallelism or the cloud provider's task invocation limit (e.g., 1000 default on AWS).

Pocket's API also accepts hints for the aggregate throughput and capacity requirements of a job, which override Pocket's heuristic estimates. This information can come from profiling. When Pocket receives a throughput hint with no capacity hint, the controller allocates capacity proportional to the job's throughput allocation. The proportion is set by the storage throughput to capacity ratio on the VMs used (e.g., `i3.2xl` instances in AWS provide 1.9 TB of capacity per ∼ 8 Gb/s of network bandwidth). Vice versa, if only a capacity hint is provided, Pocket allocates throughput based on the VM capacity:throughput ratio. In the future, we plan to allow jobs to specify their average *per-lambda* throughput and capacity requirements, as these can be more meaningful than aggregate throughput and capacity hints for a job when the number of lambdas used is subject to change.

The hints in Table 3 can be specified by application developers or provided by the application framework. For example, the framework we use to run lambda-distributed software compilation automatically infers and synthesizes a job's dependency graph [31]. Hence, this framework can provide Pocket with hints about the job's maximum degree of parallelism, for instance.

Assigning resources: Pocket translates a job's resource allocation into a resource assignment on specific storage servers by generating a *weight map* for the job. The weight map is an associative array mapping each storage server (identified by its IP address and port) to a weight from 0 to 1, which represents the fraction of a job's dataset to place on that storage server. If a storage server is assigned a weight of 1 in a job's weight map, it will store all of the job's data. The controller sends the weight map to metadata servers, which enforce the data placement policy by routing client requests to storage servers using weighted random selection based on the weights in the job's weight map.

The weight map depends on the job's resource requirements and the available cluster resources. Pocket uses an online bin-packing algorithm which first tries to fit a job's throughput, capacity and storage media allocation on active storage servers and only launches new servers if the job's requirements cannot be satisfied by

sharing resources with other jobs [67]. If a job requires more resources than are currently available, the controller launches the necessary storage nodes while the application waits for its job registration command to return. Nodes take a few seconds or minutes to launch, depending on whether a new VM is required (§6.2).

4.2 Rightsizing the Storage Cluster

In addition to rightsizing the storage allocation for each job, Pocket dynamically scales cluster resources to accommodate elastic application load for multiple jobs over time. At its core, the Pocket cluster consists of a few long-running nodes used to run the controller, the minimum number of metadata servers (one in our deployment), and the minimum number of storage servers (two in our deployment). In particular, data written with the PERSIST flag described in §3.2, which has longer lifetime, is always stored on long-running storage servers in the cluster. Beyond these persistent resources, Pocket scales resources on demand based on load. We first describe the mechanism for horizontal and vertical scaling and then discuss the policy Pocket uses to balance cluster load by carefully steering requests across servers.

Mechanisms: The controller monitors cluster resource utilization by processing heartbeats from storage and metadata servers containing their CPU, network, and storage media capacity usage. Nodes send statistics to the controller every second. The interval is configurable.

When launching a new storage server, the controller provides the IP addresses of all metadata servers that the storage server must establish connections with to join the cluster. The new storage server registers a portion of its capacity with each of these metadata servers. Metadata servers independently manage their assigned capacity and do not communicate with each other. Storage servers periodically sends heartbeats to metadata servers.

To remove a storage server, the controller blacklists the storage server by assigning it a zero weight in the weight maps of incoming jobs. This ensures that metadata servers do not steer data from new jobs to this node. The controller instructs a randomly selected metadata server to set a 'kill' flag in the heartbeat responses of the blacklisted storage server. The blacklisted storage server waits until its capacity is entirely freed, as jobs terminate and their ephemeral data are garbage collected. The storage server then terminates and releases its resources.

When the controller launches a new metadata server, the metadata server waits for new storage servers to also be launched and register their capacity. To remove a metadata server, the controller sends a 'kill' RPC to the node. The metadata server waits for all the capacity it manages to be freed, then notifies all connected storage servers to close their connections. When all connections close, the metadata server terminates. Storage servers then register their capacity that was managed by the old metadata server across new metadata servers.

In addition to horizontal scaling, the controller manages vertical scaling. When the controller observes that CPU utilization is high and additional cores are available on a node, the controller instructs the node via a heartbeat response to use additional CPU cores.

Cluster sizing policy: Pocket elastically scales the cluster using a policy that aims to maintain overall utilization for each resource type (CPU, network bandwidth, and the capacity of each storage tier) within a target range. The target utilization range can be configured separately for each resource type and managed separately for metadata servers, long-running storage servers (which store data written with the PERSIST flag set) and regular storage servers. For our deployment, we use a lower utilization threshold of 60% and a upper utilization threshold of 80% for all resource dimensions, for both the metadata and storage nodes. The range is empirically tuned and depends on the time it takes to add/remove nodes. Pocket's controller scales down the cluster by removing a storage server if overall CPU, network bandwidth *and* capacity utilization is below the lower limit of the target range. In this case, Pocket removes a storage server belonging to the tier with lowest capacity utilization. Pocket adds a storage server if overall CPU, network bandwidth *or* capacity utilization is above the upper limit of the target range. To respond to CPU load spikes or lulls, Pocket first tries to vertically scale CPU resources on metadata and storage servers before horizontally scaling the number of nodes.

Balancing load with data steering: To balance load while dynamically sizing the cluster, Pocket leverages the short-lived nature of ephemeral data and serverless jobs. As ephemeral data objects only live for tens to hundreds of seconds (see Figure 3), migrating this data to re-distribute load when nodes are added or removed has high overhead. Instead, Pocket focuses on steering data for incoming jobs across active and new storage servers joining the cluster. Pocket controls data steering by assigning specific weights for storage servers in each job's weight map. To balance load, the controller assigns higher weights to under-utilized storage servers.

The controller uses a similar approach, at a coarser granularity, to balance load across metadata servers. As noted in §3.2, the controller currently assigns each job to one metadata server. The controller estimates the load a job will impose on a metadata server based on its throughput and capacity allocation. Combining this estimate with metadata server resource utilization statistics, the controller selects a metadata server to use for an incoming job such that the predicted metadata server resource utilization remains within the target range.

5 Implementation

Controller: Pocket's controller, implemented in Python, leverages the Kubernetes container orchestration system to launch and tear down metadata and storage servers, running in separate Docker containers [7]. The controller uses Kubernetes Operations (kops) to spin up and down virtual machines that run containers [6]. As explained in §4.2, Pocket rightsizes cluster resources to maintain a target utilization range. We implement a resource monitoring daemon in Python which runs on each node, sending CPU and network utilization statistics to the controller every second. Metadata servers also send storage tier capacity utilization statistics. We empirically tune the target utilization range based on node startup time. For example, we use a conservative target utilization range when the controller needs to launch new VMs compared to when VMs are running and the controller simply launches containers.

Metadata management: We implement Pocket's metadata and storage server architecture on top of the Apache Crail distributed data store [2, 71]. Crail is designed for low latency, high throughput storage of arbitrarily sized data with low durability requirements. Crail provides a unified namespace across a set of heterogeneous storage resources distributed in a cluster. Its modular architecture separates the data and metadata plane and supports pluggable storage tier and RPC library implementations. While Crail is originally designed for RDMA networks, we implement a TCP-based RPC library for Pocket since RDMA is not readily available in public clouds. Like Crail, Pocket's metadata servers are implemented in Java. Each metadata server logs its metadata operations to a file on a shared NFS mount point, such that the log can be accessed and replayed by a new metadata server in case a metadata server fails.

Storage tiers: We implement three different storage tiers for Pocket. The first is a DRAM tier implemented in Java, using NIO APIs to efficiently serve requests from clients over TCP connections. The second tier uses NVMe Flash storage. We implement Pocket's NVMe storage servers on top of ReFlex, a system that allows clients (i.e., lambdas) to access Flash over commodity Ethernet networks with high performance [47]. ReFlex is implemented in C and leverages Intel's DPDK [43] and SPDK [44] libraries to directly access network and NVMe device queues from userspace. ReFlex uses a polling-based execution model to efficiently process network storage requests over TCP. The system also uses a quality of service (QoS) aware scheduler to manage read/write interference on Flash and provide predictable performance to clients. The third tier we implement is a generic block storage tier that allows Pocket to use any block storage device (e.g., HDD or SATA/SAS SSD) via a standard kernel device driver. Similar to ReFlex, this tier is implemented in C and uses DPDK for efficient, userspace networking. However, instead of using SPDK to access NVMe Flash devices from userspace, this tier uses the Linux libaio library to submit asynchronous block storage requests to a kernel block device driver. Leveraging userspace APIs for the Pocket NVMe and generic block device tiers allows us to increase performance and resource efficiency. For example, ReFlex can process up to 11× more requests per core than a conventional Linux network-storage stack [47].

Pocket server	EC2 server	DRAM (GB)	Storage (TB)	Network (Gb/s)	$ / hr
Controller	`m5.xl`	16	0	∼8	0.192
Metadata	`m5.xl`	16	0	∼8	0.192
DRAM	`r4.2xl`	61	0	∼8	0.532
NVMe	`i3.2xl`	61	1.9	∼8	0.624
SSD	`i2.2xl`	61	1.6	$\lesssim 2$	1.705[1]
HDD	`h1.2xl`	32	2	∼8	0.468

Table 4: Type and cost of EC2 VMs used for Pocket

Client library: Since the serverless applications we use are written in Python, we implement Pocket's application interface (Table 2) as a Python client library. The core of the library is implemented in C++ to optimize performance. We use `Boost` to wrap the code into a Python library. The library internally manages TCP connections with metadata and storage servers.

6 Evaluation

6.1 Methodology

We deploy Pocket on Amazon Web Service (AWS). We use EC2 instances to run Pocket storage, metadata, and controller nodes. We use four different kinds of storage media: DRAM, NVMe-based Flash, SATA/SAS-based Flash (which we refer to as SSD), and HDD. DRAM servers run on `r4.2xl` instances, NVMe Flash servers run on `i3.2xl` instances, SSD servers run on `i2.2xl` instances, and HDD servers run on `h1.2xl` instances. We choose the instance families based on their local storage media, shown in Table 4. We choose the VM size to provide a good balance of network bandwidth and storage capacity for the serverless applications we study.

We run Pocket storage and metadata servers as containers on EC2 VMs, orchestrated with Kubernetes v1.9. We use AWS Lambda as our serverless computing platform. We enable lambdas to access Pocket EC2 nodes by deploying them in the same virtual private cloud (VPC).

[1] The cost of the `i2` instance is particularly high since it is an old generation instance that is being phased out by AWS and replaced by the newer generation `i3` instances with NVMe Flash devices.

We configure lambdas with 3 GB of memory. Amazon allocates lambda compute resources proportional to memory resources [18]. We compare Pocket's performance and cost-efficiency to ElastiCache Redis (cluster-mode enabled) and Amazon S3 [13, 51, 16]. We present results from experiments conducted in April 2018.

We study three different serverless analytics applications, described below. The applications differ in their degree of parallelism, ephemeral object size distribution (Figure 2), and throughput requirements.

Video analytics: We use the Thousand Island Scanner (THIS) for distributed video processing [63]. Lambdas in the first stage read compressed video frame batches, decode, and write the decoded frames to ephemeral storage. Each lambda fetches a 250 MB decoder executable from S3 as it does not fit in the AWS Lambda deployment package. Each first stage lambda then launches second stage lambdas, which read decoded frames from ephemeral storage, compute a MXNET deep learning classification algorithm and output an object detection result. We use a 25 minute video with 40K 1080p frames. We tune the batch size for each stage to minimize the job's end-to-end execution time; the first stage consists of 160 lambdas while the second has 305 lambdas.

MapReduce Sort: We implement a MapReduce sort application on AWS Lambda, similar to PyWren [45]. Map lambdas fetch input files from long-term storage (we use S3) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data and upload output files to long-term storage. We run a 100 GB sort, which generates 100 GB of ephemeral data. We run the job with 250, 500, and 1000 lambdas.

Distributed software compilation (λ-cc): We use gg to infer software build dependency trees and invoke lambdas to compile source code with high parallelism [4, 31]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links), and writes its output to ephemeral storage, including the final executable for the user to download. We present results for compiling the `cmake` project source code. This build job has a maximum inherent parallelism of 650 tasks and generates a total of 850 MB ephemeral data. Object size ranges from 10s of bytes to MBs, as shown in Figure 2.

6.2 Microbenchmarks

Storage request latency: Figure 5 compares the 1 KB request latency of S3, Redis, and various Pocket storage tiers measured from a lambda client. Pocket-DRAM, Pocket-NVMe and Redis latency is below 540 μs, which is over 45$\times$ faster than S3. The latency of the Pocket-SSD and Pocket-HDD tiers is higher due to higher storage media access times. Pocket-HDD `get` latency is higher than `put` latency since lambdas issue random reads while writes are sequential; the metadata server routes writes to sequential logical block addresses. Pocket-DRAM has higher latency than Redis mainly due to the metadata lookup RPC, which takes 140 μs. While Redis cluster clients simply hash keys to Redis nodes, Pocket clients must contact a metadata server. While this extra RPC increases request latency, it allows Pocket to optimize data placement per job and dynamically scale the cluster without redistributing data across nodes.

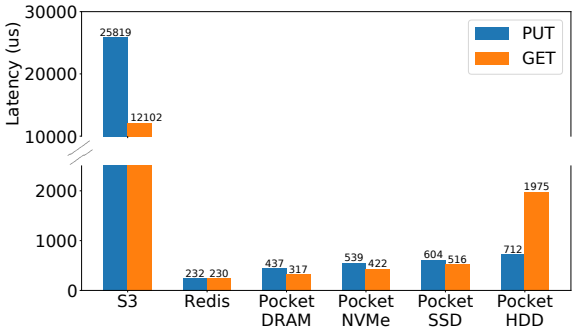


Figure 5: Unloaded latency for 1KB access from lambda

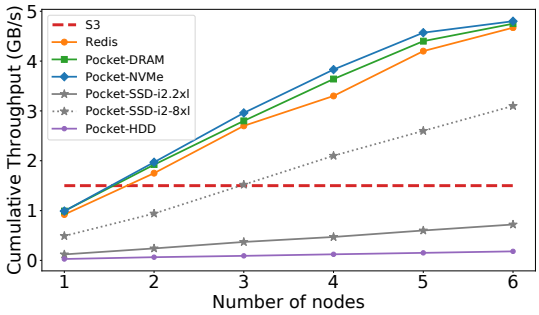


Figure 6: Total GB/s for 1MB requests from 100 lambdas

Storage request throughput: We measure the `get` throughput of S3, Redis (`cache.r4.2xl`) and various Pocket storage tiers by issuing 1 MB requests from 100 concurrent lambdas. In Figure 6, we sweep the number of nodes in the Redis and Pocket clusters and compare the cumulative throughput to that achieved with S3. Pocket-DRAM, Pocket-NVMe and Redis all achieve similar throughput. With a single node, the bottleneck is the 1 GB/s VM network bandwidth. With two nodes, Pocket's DRAM and NVMe tiers achieve higher throughput than S3. Pocket's SSD and HDD tiers have significantly lower throughput. The HDD tier is limited by the 40 MB/s random access bandwidth of the disk on each node. The SSD tier is limited by poor networking (less than $\sim$2 Gb/s) on the old generation `i2.2xl` instances. Hence, we also plot the throughput using `i2.8xl` instances which have 10 Gb/s networking. The

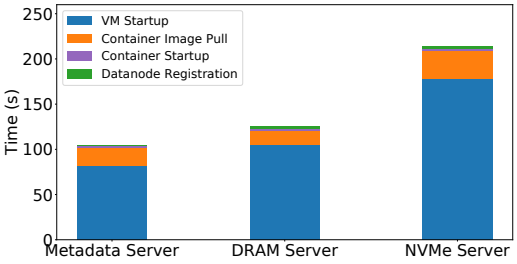


Figure 7: Node startup time breakdown

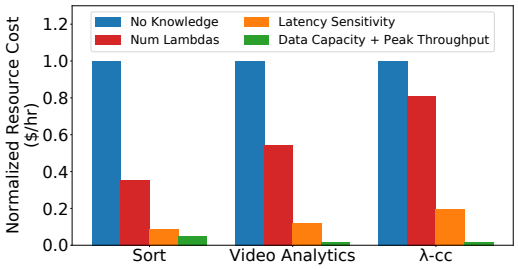


Figure 8: Pocket leverages cumulative hints about job characteristics to allocate resources cost-efficiently.

bottleneck becomes the 500 MB/s throughput limit of the SATA/SAS SSD.

We focus the rest of our evaluation of Pocket on the DRAM and NVMe Flash tiers as they demand the highest data plane software efficiency due to the technology's low latency and high throughput. We also find that in our AWS deployment, the DRAM and NVMe tiers offer significantly higher performance-cost efficiency compared to the HDD and SSD tiers. For example, NVMe Flash servers, which run on on `i3.2xl` instances, provide 1 GB/s per 1900 GB capacity at a cost of $0.624/hour. Meanwhile, HDD servers, which run on `h1.2xl` instances, provide only 40 MB/s per 2000 GB capacity at a cost of $0.468/hour. Thus, the NVMe tier offers 19.7× higher throughput per GB per dollar.

Metadata throughput: We measure the number of metadata operations that a metadata server can handle per second. A single core metadata server on the `m5.xl` instance supports up to 90K operations per second and up to 175K operations per second with four cores. The peak metadata request rate we observe for the serverless analytics applications we study is 75 operations per second per lambda. Hence, a multi-core metadata server can support jobs with thousands of lambdas.

Adding/removing servers: Since Pocket runs in containers on EC2 nodes, we measure the time it takes to launch a VM, pull the container image, and launch the container. Pocket storage servers must also register their storage capacity with metadata servers to join the cluster. Figure 7 shows the time breakdown. VM launch time varies across EC2 instance types. The container image for the metadata server and DRAM server has a compressed size of 249 MB while the Pocket-NVMe compressed container image is 540 MB due to dependencies for DPDK and SPDK to run ReFlex. The image pull time depends on the VM's network bandwidth. The VM launch time and container image pull time only need to be done once when the VM is first started. Once the VM is warm, meaning the image is available locally, starting and stopping containers takes only a few seconds. The time to terminate a VM is tens of seconds.

6.3 Rightsizing Resource Allocations

We now evaluate Pocket with the three different serverless applications described in §6.1.

Rightsizing with application hints: Figure 8 shows how Pocket leverages user hints to make cost-effective resource allocations, assuming each hint is provided in addition to the previous ones. With no knowledge of application requirements, Pocket defaults to a policy that spreads data for a job across a default allocation of 50 nodes, filling DRAM first, then Flash. With knowledge of the maximum number of concurrent lambdas (250, 160, and 650 for the sort, video analytics and λ-cc jobs, respectively), Pocket allocates lower aggregate throughput than the default allocation while maintaining similar job execution time (within 4% of the execution time achieved with the default allocation). Furthermore, these jobs are not sensitive to latency; the sort job and the first stage of the video analytics job are throughput intensive while λ-cc and the second stage of the video analytics job are compute limited. The orange bars in Figure 8 show the cost savings of using NVMe Flash as opposed to DRAM when the latency insensitivity hint is provided for these jobs. The green bar shows the relative resource allocation cost when applications provide explicit hints for their capacity and peak throughput requirements; such hints can be obtained from a profiling run. Across all scenarios, each job's execution time remains within 4% of its execution time with the default resource allocation.

Reclaiming capacity using hints: Figure 9 shows the capacity used over time for the video analytics job, with and without data lifetime management hints. All ephemeral data in this application is written and read only once, since each first stage lambda writes ephemeral data destined to a single second stage lambda. Hence for all `get` operations, this job can make use of the DELETE hint which informs Pocket to promptly garbage collect an object as soon as it has been read. By default, when the DELETE hint is not specified, Pocket waits until the job deregisters to delete the job's data. The job in Figure 9

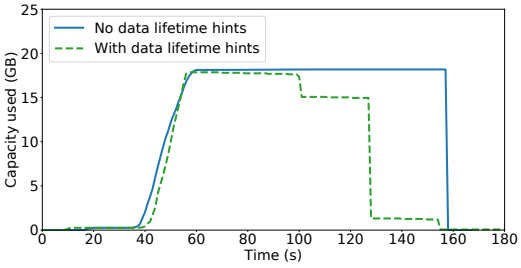


Figure 9: Example of using the DELETE hint for `get` operations in a video analytics job, enabling Pocket to readily reclaim capacity by deleting objects after they have been read versus waiting for the job to complete.

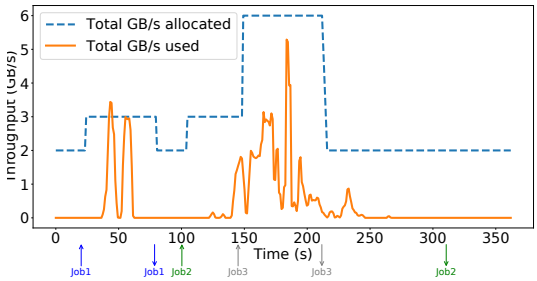


Figure 10: Pocket's controller dynamically scales cluster resources to meet I/O requirements as jobs come and go.

completes at the 158 second mark. We show that leveraging the DELETE hint allows Pocket to reclaim capacity more promptly, making more efficient use of resources as this capacity can be offered to other jobs.

Rightsizing cluster size: Elastic and automatic resource scaling is a key property of Pocket. Figure 10 shows how Pocket scales cluster resources as multiple jobs register and deregister with the controller. Job registration and deregistration times are indicated by upwards and downwards arrows along the x-axis, respectively. In this experiment, we assume Pocket receives capacity and throughput hints for each job's requirements. The first job is a 10 GB sort application requesting 3 GB/s, the second job is a video analytics application requesting 2.5 GB/s and the third job is a different invocation of a 10 GB sort also requesting 3 GB/s. Each storage server provides 1 GB/s. We use a minimum of two storage servers in the cluster. We provision seven VMs for this experiment and ensure that storage server containers are locally available, such that when the controller launches new storage servers, only container startup and capacity registration time is included.

Figure 10 shows that Pocket quickly and effectively scales the allocated storage bandwidth (dotted line) to meet application throughput demands (solid line). The spike surpassing the allocated throughput is due to a short burst in EC2 VM network bandwidth. The VMs provide 'up to 10 Gb/s', but since we typically observe a $\sim$8 Gb/s bandwidth limit in practice, the controller allocates throughput assuming each node provides 8 Gb/s. As the controller rightsizes resources for each job, job execution time stays within 5% of its execution time when running on 50 nodes, the conservative default resource allocation. If the controller had to spin up new VMs to accommodate a job's requirements instead of just launching containers, the job's start time would be delayed by up to 215 seconds (see EC2 NVMe server startup time in Figure 7) since the `register_job` call blocks until the required storage servers are available.

6.4 Comparison to S3 and Redis

Job execution time: Figure 11 plots the per-lambda execution time breakdown for the MapReduce 100 GB sort job, run with 250, 500, and 1000 concurrent lambdas. The purple bars show the time spent fetching original input data and writing final output data to S3 while the blue bars compare the time for ephemeral data I/O with S3, Redis and Pocket-NVMe. S3 does not support sufficient request rates when the job is run with 500 or more lambdas. S3 returns errors, advising to reduce the I/O rate. Pocket provides similar throughput to Redis, however since the application is not sensitive to latency, Pocket uses NVMe Flash instead of DRAM to reduce cost.

Similarly, for the video analytics job, we observe that Pocket-NVMe achieves the same performance as Redis. However, using S3 for the video analytics job increases the average time spent on ephemeral I/O by each lambda in the first stage (video decoding) by 3.2$\times$ and 4.1$\times$ for lambdas in the second stage (MXNET classification), compared to using Pocket or Redis.

The performance of the distributed compilation job (λ-cc cmake) is limited by lambda CPU resources [49]. A software build job has inherently limited parallelism; early-stage lambdas compile independent files in parallel, however lambdas responsible for archiving and linking are serialized as they depend on the outputs of the early-stage lambdas. We observe that the early-stage lambdas are compute-bound on current serverless infrastructure. Although using Pocket or Redis reduces the fraction of time each lambda spend on ephemeral I/O, the overall execution time for this job remains the same as when using S3 for ephemeral storage, since the bottleneck is dependencies on compute-bound lambdas.

Cost analysis: Table 4 shows the hourly cost of running Pocket nodes on EC2 VMs in April 2018. Our minimum size Pocket cluster, consisting of one controller node, one metadata server and two `i3.2xl` storage nodes costs $1.632 per hour on EC2. However, Pocket's fixed cost can be amortized as the system is designed to support multiple concurrent jobs from one or more tenants. We intend for Pocket to be operated by a cloud provider

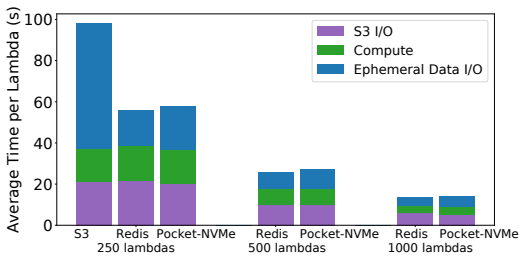


Figure 11: Execution time breakdown of 100GB sort.

Job	S3	Redis	Pocket
100 GB sort	0.05126	5.320	2.1648
Video analytics	0.00034	1.596	0.6483
λ-cc cmake	0.00005	1.596	0.6480

Table 5: Hourly ephemeral storage cost (in USD)

and offered as a storage service with a pay-what-you-use cost model for users, similar to the cost model of serverless computing platforms. Hence, for our cost analysis, we derive fine-grain resource costs, such as the cost of a CPU core and the cost of storage per GB, using AWS EC2 instance pricing. For example, we calculate NVMe Flash $/GB by taking the difference between `i3.2xl` and `r4.2xl` instance costs (since these VMs have the same CPU and DRAM configurations but `i3.2xl` includes a 1900 GB NVMe drive) and dividing by the GB capacity of the `i3.2xl` NVMe drive.

Using this fine-grain resource pricing model for Pocket, Table 5 compares the cost of running the 100 GB sort, video analytics and distributed compilation jobs with S3, ElastiCache Redis, and Pocket-NVMe. We use reduced redundancy pricing for S3 and assume the GB-month cost is charged hourly [15]. We base Redis costs on the price of entire VMs, not only the resources consumed, since ElastiCache Redis clusters are managed by individual users rather than cloud providers. Pocket achieves the same performance as Redis for all three jobs while saving 59% in cost. S3 is still orders of magnitude cheaper. However, S3's cloud provider based cost is not a fair comparison to the cloud user based cost model we use for Pocket and Redis. Furthermore, while the λ-cc job has similar performance with Pocket, Redis and S3 due to a lambda compute bottleneck, the video analytics and sort job execution time is 40 to 65% higher with S3.

7 Discussion

Choice of API: Pocket's simple `get/put` interface provides sufficient functionality for the applications we studied. Lambdas in these jobs consume entire data objects that they read and they do not require updating or appending files. However, POSIX-like I/O semantics for appending or accessing parts of objects could benefit other applications. Pocket's `get/put` API is implemented on top of Apache Crail's append-only stream abstraction which allows clients to read at file offsets and append to files with single-writer semantics [3]. Thus, Pocket's API could easily be modified to expose Crail's I/O semantics. Other operators such as filters or multi-gets could also help optimize the number of RPCs and bytes transferred. The right choice of API for ephemeral storage remains an open question.

Security: Pocket uses access control to secure applications in a multi-tenant environment. To prevent malicious users from accessing other tenants' data, metadata servers issue single-use certificates to clients which are verified at storage servers. An I/O request that is not accompanied with a valid certificate is denied. Clients communicate with metadata servers over SSL to protect against man in the middle attacks. Users set cloud network security rules to prevent TCP traffic snooping on connections between lambdas and storage servers. Alternatively, users can encrypt their data. Pocket does not currently prevent jobs from issuing higher load than specified in job registration hints. Request throttling can be implemented at metadata servers to mitigate interference when a job tries to exceed its allocation.

Learning job characteristics: Pocket currently relies on user or application framework hints to cost-effectively rightsize resource allocations for a job. Currently, Pocket does not autonomously learn application properties. Since users may repeatedly run jobs on different datasets, as many data analytics and modern machine learning jobs are recurring [55], Pocket's controller can maintain statistics about previous invocations of a job and use this information combined with machine learning techniques to rightsize resource allocations for future runs [48, 10]. We plan to explore this in future work.

Applicability to other cloud platforms: While we evaluate Pocket on the AWS cloud platform, the system addresses a real problem applicable across all cloud providers as no available platform provides an optimized way for serverless tasks to exchange ephemeral data. Pocket's performance will vary with network and storage capabilities of different infrastructure. For example, if a low latency network is available, the DRAM storage tier provides significantly lower latency than the NVMe tier. Such variations emphasize the need for a control plane to automate resource allocation and data placement.

Applicability to other cloud workloads: Though we presented Pocket in the context of ephemeral data sharing in serverless analytics, Pocket can also be used for other applications that require distributed, scalable temporary storage. For instance, Google's Cloud Dataflow, a fully-managed data processing service for streaming and batch data analytics pipelines, implements the shuf-

fle operator – used for transforms such as `GroupByKey` – as part of its service backend [35]. Pocket can serve as fast, elastic storage for the intermediate data generated by shuffle operations in this kind of service.

Reducing cost with resource harvesting: Cloud jobs are commonly over-provisioned in terms CPU, DRAM, network, and storage resources due to the difficulty of rigthsizing general jobs and the need to accommodate diurnal load patterns and unexpected load spikes. The result is significant capacity underutilization at the cluster level [21, 74, 29]. Recent work has shown that the plethora of allocated but temporarily unused resources provide a stable substrate that can be used to run analytics job [22, 79]. We can similarly leverage harvested resources to dramatically reduce the total cost of running Pocket. Pocket's storage servers are particularly well suited to run on temporarily idle resource as ephemeral data has short lifetime and low durability requirements.

8 Related Work

Elastic resource scaling: Various reactive [34], predictive [25, 50, 65, 30, 62, 72, 75] and hybrid [24, 41, 33, 60] approaches have been proposed to automatically scale resources based on demand [64, 61]. Muse takes an economic approach, allocating resources to their most efficient use based on a utility function that estimates the impact of resource allocations on job performance [23]. Pocket provisions resources upfront for a job based on hints and conservative heuristics while using a reactive approach to adjust cluster resources over time as jobs enter and leave the system. Pocket's reactive scaling is similar to Horizontal Pod autoscaling in Kubernetes which collects multidimensional metrics and adjusts resources based on utilization ratios [5]. Petal [52] and the controller by Lim et al. [54] propose data re-balancing strategies in elastic storage clusters while Pocket avoids redistributing short-lived data due to the high overhead. CloudScale [68], Elastisizer [40], CherryPick [11], and other systems [73, 77, 48] take an application-centric view to rightsize a job at the coarse granularity of traditional VMs as opposed to determining fine-grain storage requirements. Nevertheless, the proposed cost and performance modeling approaches can also be applied to Pocket to autonomously learn job resource preferences.

Intelligent data placement: Mirador is a dynamic storage service that optimizes data placement for performance, efficiency, and safety [76]. Mirador focuses on long-running jobs (minutes to hours), while Pocket targets short-term (seconds to minutes) ephemeral storage. Tuba manages geo-replicated storage and, similar to Pocket, optimizes data placement based on performance and cost constraints received from applications [20]. Extent-based Dynamic Tiering (EDT) uses access pattern simulations and monitoring to find a cost-efficient storage solution for a workload across multiple storage tiers [38]. The access pattern of ephemeral data is often simple (e.g., write-once-read-once) and the data is short-lived, hence it is not worth migrating between tiers. Multiple systems make storage configuration recommendation based on workload traces [19, 70, 9, 59, 12]. Given I/O traces for a job, Pocket could apply similar techniques to assign resources when a job registers.

Fully managed data warehousing: Cloud providers offer fully managed infrastructure for querying large amounts of structured data with high parallelism and elasticity. Examples include Amazon Redshift [14], Google BigQuery [37], Azure SQL Data Warehouse [58], and Snowflake [26]. These systems are designed to support relational queries and high data durability, while Pocket is designed for elastic, fast, and fully managed storage of data with low durability requirements. However, a cloud data warehouse like Snowflake, which currently stores temporary data generated by query operators on local disk or S3, could leverage Pocket to improve elasticity and resource utilization.

9 Conclusion

General-purpose analytics on serverless infrastructure presents unique opportunities and challenges for performance, elasticity and resource efficiency. We analyzed challenges associated with efficient data sharing and presented Pocket, an ephemeral data store for serverless analytics. In a similar spirit to serverless computing, Pocket aims to provide a highly elastic, cost-effective, and fine-grained storage solution for analytics workloads. Pocket achieves these goals using a strict separation of responsibilities for control, metadata, and data management. To the best of our knowledge, Pocket is the first system designed specifically for ephemeral data sharing in serverless analytics workloads. Our evaluation on AWS demonstrates that Pocket offers high performance data access for arbitrary size data sets, combined with automatic fine-grain scaling, self management and cost effective data placement across multiple storage tiers.

Acknowledgements

We thank our shepherd, Hakim Weatherspoon, and the anonymous OSDI reviewers for their helpful feedback. We thank Qian Li, Francisco Romero, and Sadjad Fouladi for insightful technical discussions. This work is supported by the Stanford Platform Lab, Samsung, and Huawei. Ana Klimovic is supported by a Stanford Graduate Fellowship. Yawen Wang is supported by a Stanford Electrical Engineering Department Fellowship.

References

[1] Apache CouchDB. http://couchdb.apache.org, 2018.

[2] Apache Crail (incubating). http://crail.incubator.apache.org, 2018.

[3] Crail Storage Performance – Part I: DRAM. http://crail.incubator.apache.org/blog/2017/08/crail-memory.html, 2018.

[4] gg: The Stanford Builder. https://github.com/stanfordsnr/gg, 2018.

[5] Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, 2018.

[6] Kubernetes operations (kops). https://github.com/kubernetes/kops, 2018.

[7] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io, 2018.

[8] Memcached – a distributed memory object caching system. https://memcached.org, 2018.

[9] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proc. of the USENIX Annual Technical Conference* (2013), ATC'13, pp. 91–102.

[10] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), pp. 469–482.

[11] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)* (2017), pp. 469–482.

[12] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[13] AMAZON. Amazon ElastiCache. https://aws.amazon.com/elasticache, 2018.

[14] AMAZON. Amazon redshift. https://aws.amazon.com/redshift, 2018.

[15] AMAZON. Amazon S3 reduced redundancy storage. https://aws.amazon.com/s3/reduced-redundancy, 2018.

[16] AMAZON. Amazon simple storage service. https://aws.amazon.com/s3, 2018.

[17] AMAZON. AWS lambda. https://aws.amazon.com/lambda, 2018.

[18] AMAZON. AWS lambda limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2018.

[19] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST'02, pp. 13–13.

[20] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 367–381.

[21] BARROSO, L. A., CLIDARAS, J., AND HLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[22] CARVALHO, M., CIRNE, W., BRASILEIRO, F., AND WILKES, J. Long-term SLOs for reclaimed cloud computing resources. In *Proc. of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 20:1–20:13.

[23] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A. M., AND DOYLE, R. P. Managing energy and server resources in hosting centers. In *Proc. of the Eighteenth ACM Symposium on Operating Systems Principles* (2001), SOSP '01, pp. 103–116.

[24] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), NSDI'08, pp. 337–350.

[25] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proc. of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 153–167.

[26] DAGEVILLE, B., CRUANES, T., ZUKOWSKI, M., ANTONOV, V., AVANES, A., BOCK, J., CLAYBAUGH, J., ENGOVATOV, D., HENTSCHEL, M., HUANG, J., LEE, A. W., MOTIVALA, A., MUNIR, A. Q., PELLEY, S., POVINEC, P., RAHN, G., TRIANTAFYLLIS, S., AND UNTERBRUNNER, P. The snowflake elastic data warehouse. In *Proc. of the International Conference on Management of Data* (2016), SIGMOD '16, pp. 215–226.

[27] DATABRICKS. Databricks serverless: Next generation resource management for Apache Spark. `https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html`, 2017.

[28] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07, pp. 205–220.

[29] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware cluster management. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 127–144.

[30] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., AND VAHDAT, A. M. Model-based resource provisioning in a web service utility. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems* (2003), USITS'03, pp. 5–5.

[31] FOULADI, S., ITER, D., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint). `http://stanford.edu/~sadjad/gg-paper.pdf`.

[32] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI'17, pp. 363–376.

[33] GANDHI, A., CHEN, Y., GMACH, D., ARLITT, M., AND MARWAH, M. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proc. of the 2011 International Green Computing Conference and Workshops* (2011), IGCC '11, pp. 1–8.

[34] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012), 14:1–14:26.

[35] GOOGLE. Introducing Cloud Dataflow Shuffle: For up to 5x performance improvement in data analytic pipelines. `https://cloud.google.com/blog/products/gcp/introducing-cloud-dataflow-shuffle-for-up-to-5x-performance-improvement-in-data-analytic-pipelines`, 2017.

[36] GOOGLE. Cloud functions. `https://cloud.google.com/functions`, 2018.

[37] GOOGLE. Google bigquery. `https://cloud.google.com/bigquery`, 2018.

[38] GUERRA, J., PUCHA, H., GLIDER, J., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies* (2011), FAST'11, pp. 20–20.

[39] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 75–88.

[40] HERODOTOU, H., DONG, F., AND BABU, S. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11, pp. 18:1–18:14.

[41] HORVATH, T., AND SKADRON, K. Multi-mode energy management for multi-tier server clusters. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques* (2008), PACT '08, pp. 270–279.

[42] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Proc. of the USENIX Conference on Annual Technical Conference* (2012), ATC'12, pp. 2–2.

[43] INTEL CORP. Dataplane Performance Development Kit. `https://dpdk.org`, 2018.

[44] INTEL CORP. Storage Performance Development Kit. `https://01.org/spdk`, 2018.

[45] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 445–451.

[46] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads. In *Proc. of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, pp. 20–20.

[47] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 345–359.

[48] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proc. of the USENIX Annual Technical Conference (ATC'18)* (2018), pp. 759–773.

[49] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *Proc. of the USENIX Annual Technical Conference (ATC'18)* (2018), pp. 789–794.

[50] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. H. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of the First ACM SIGCOMM Workshop on Green Networking* (2010), Green Networking '10, pp. 15–22.

[51] LABS, R. Redis. `https://redis.io`, 2018.

[52] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), ASPLOS VII, pp. 84–92.

[53] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 6:1–6:15.

[54] LIM, H. C., BABU, S., AND CHASE, J. S. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing* (2010), ICAC '10, pp. 1–10.

[55] MASHAYEKHI, O., QU, H., SHAH, C., AND LEVIS, P. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2017), USENIX ATC '17, USENIX Association, pp. 513–526.

[56] MICROSOFT. Azure functions. `https://azure.microsoft.com/en-us/services/functions`, 2018.

[57] MICROSOFT AZURE. Azure redis cache. `https://azure.microsoft.com/en-us/services/cache`, 2018.

[58] MICROSOFT AZURE. SQL data warehouse. `https://azure.microsoft.com/en-us/services/sql-data-warehouse`, 2018.

[59] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to ssds: Analysis of tradeoffs. In *Proc. of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, pp. 145–158.

[60] NETFLIX. Scryer: Netflixs predictive auto scaling engine. `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`, 2013.

[61] NETTO, M. A. S., CARDONHA, C., CUNHA, R. L. F., AND ASSUNCAO, M. D. Evaluating auto-scaling strategies for cloud computing environments. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems* (2014), MASCOTS '14, pp. 187–196.

[62] NGUYEN, H., SHEN, Z., GU, X., SUBBIAH, S., AND WILKES, J. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the 10th International Conference on Autonomic Computing* (2013), ICAC'13, pp. 69–82.

[63] QIAN LI, JAMES HONG, D. D. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. `https://github.com/qianl15/this`, 2018.

[64] QU, C., CALHEIROS, R. N., AND BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys 51*, 4 (July 2018), 73:1–73:33.

[65] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. of the 2011 IEEE 4th International Conference on Cloud Computing* (2011), CLOUD '11, pp. 500–507.

[66] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: novel erasure codes for big data. In *Proc. of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, pp. 325–336.

[67] SEIDEN, S. S. On the online bin packing problem. *J. ACM 49*, 5 (Sept. 2002), 640–671.

[68] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11, pp. 5:1–5:14.

[69] SRINIVASAN, V., BULKOWSKI, B., CHU, W.-L., SAYYAPARAJU, S., GOODING, A., IYER, R., SHINDE, A., AND LOPATIC, T. Aerospike: Architecture of a real-time operational DBMS. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1389–1400.

[70] STRUNK, J. D., THERESKA, E., FALOUTSOS, C., AND GANGER, G. R. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA* (2008), pp. 313–328.

[71] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin 40*, 1 (2017), 38–49.

[72] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *Proc. of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2005), SIGMETRICS '05, pp. 291–302.

[73] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), pp. 363–378.

[74] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proc. of the European Conference on Computer Systems* (Bordeaux, France, 2015), EuroSys'15.

[75] WAJAHAT, M., GANDHI, A., KARVE, A., AND KOCHUT, A. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)* (Nov 2016), pp. 1–8.

[76] WIRES, J., AND WARFIELD, A. Mirador: An active control plane for datacenter storage. In *Proc. of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17, pp. 213–228.

[77] YADWADKAR, N. J., HARIHARAN, B., GONZALEZ, J. E., SMITH, B., AND KATZ, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 452–465.

[78] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation* (2012), NSDI'12, pp. 15–28.

[79] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, pp. 755–770.

Sharding the Shards: Managing Datastore Locality at Scale with Akkio

Muthukaruppan Annamalai,[†] Kaushik Ravichandran,[†] Harish Srinivas,[†] Igor Zinkovsky,[†]
Luning Pan,[†] Tony Savor,[†] David Nagle[†] and Michael Stumm[‡,†]

[†]Facebook, 1 Hacker Way, Menlo Park, CA USA 94025
{muthu,kaushikr,harishs,igorzi,luningp,tsavor,dfnagle}@fb.com

[‡]Dept. Electrical and Computer Engineering, University of Toronto, Canada M5S 3G4
stumm@eecg.toronto.edu

Abstract

Akkio is a locality management service layered between client applications and distributed datastore systems. It determines how and when to migrate data to reduce response times and resource usage. Akkio primarily targets multi-datacenter geo-distributed datastore systems. Its design was motivated by the observation that many of Facebook's frequently accessed datasets have low R/W ratios that are not well served by distributed caches or full replication. Akkio's unit of migration is called a *μ-shard*. Each μ-shard is designed to contain related data with some degree of access locality. At Facebook, μ-shards have become a first-class abstraction.

Akkio went into production at Facebook in 2014, and it currently manages ∼100PB of data. Measurements from our production environment show that Akkio reduces access latencies by up to 50%, cross-datacenter traffic by up to 50%, and storage footprint by up to 40% compared to reasonable alternatives. Akkio is scalable: it can support trillions of μ-shards and process many 10's of millions of data access requests per second. And it is portable: it currently supports five datastore systems.

1 Introduction

This paper regards the management of data access locality in large distributed datastore systems. Our work in this area was initially motivated by our aim to reduce service response times and resource usage in our cloud environment which operates globally and at scale: the computing and storage resources are located in multiple geo-distributed datacenters, hundreds of petabytes of data must be available for access, data accesses occur at the rate of many tens of millions per second, and the location from which any data item is accessed changes dynamically over time. Many organizations are increasingly faced with some, if not all, of these aspects, as they target a growing user base around the world. Indeed, geo-distributed systems are becoming increasingly prevalent and important, as witnessed by Spanner Cloud and CockroachDB, two cloud-based geo-distributed datastore systems available to any organization [17, 38].

Managing data access locality[1] in geo-distributed systems is important because doing so can significantly improve data access latencies, given that intra-datacenter communication latencies are two orders of magnitude smaller than cross-datacenter communication latencies; e.g., 1ms vs. 100ms. Locality management can also significantly reduce cross-datacenter bandwidth usage, which is important because the bandwidth available between datacenters is often limited (§2.1), potentially leading to communication bottlenecks and attendantly higher communication latencies. Managing locality is all the more challenging when considering that access patterns can change geographically over time; particularly, when shifting workload from one datacenter operating at high utilization (e.g., during its day) to another operating at low utilization (e.g., its night) (§2.2).

We argue that explicit data migration is a necessary mechanism for managing data access locality in geo-distributed environments, because existing alternatives have serious drawbacks in many scenarios. For instance, distributed caches can be used to improve data read access locality. However, because misses often incur remote communications, these caches require extremely high cache hit rates to be effective, thus demanding significant hardware infrastructure. Further, distributed caches do not typically offer strong consistency (§2.4). Another alternative is to fully replicate data with a copy in each datacenter to allow for (fast) localized read accesses. However, as the number of datacenters increases, storage overhead becomes exorbitant with large amounts of data, and also write overheads increase significantly, as all replicas need to be updated on each write (§2.1). At Facebook, many of the heavily accessed datasets have

[1] Our use of the term *locality* should not be confounded with the term *localization*; the solution we propose here is not suitable for segregating data by region.

relatively low read-write ratios (§2.3), so full replication would consume excessive cross-datacenter bandwidth. A third alternative is function shipping. But this can also be ineffective, as it may still result in significant cross-datacenter communications, the target datacenter may be operating at peak capacity, or the required data may be located in multiple datacenters.

Akkio. In this paper we present ***Akkio***,[2] a locality management service for distributed datastore systems whose aim is to improve data access response times and to reduce cross-datacenter bandwidth usage as well as the total amount of storage capacity needed. Akkio is layered between client applications servicing client requests and the distributed datastore systems used natively by the client applications. It decides in which datacenter to place and how and when to migrate data, and it does so in a way that is transparent to its clients and the underlying datastore system.[3] It helps direct each data access to where the target data is located, and it tracks each access to be able to make appropriate placement decisions. Akkio has been in production use at Facebook since 2014 and thus operates at scale: it currently manages over 100PB of data and processes many tens of millions of data accesses per second (despite Akkio not being suitable many of Facebook's datasets).

µ-shards. Having migration as the basis for providing data access locality raises the question: what is the right granularity for migrating data? A ubiquitous method in distributed datastore systems is to partition the data into *shards* using key ranges or key hashing [26, 37]. Shards serve as the unit for replication, failure recovery, and load balancing (e.g., upon detection of query or storage load imbalances, shards are migrated from one node to another to rebalance the load). Each shard is on the order of one to a few tens of gigabytes, is assigned in its entirety to a node, and multiple shards (10s – 100s) are assigned to a node. Shard sizes are set by the datastore administrator to balance (*i*) the amount of metadata needed to manage the shards with (*ii*) effectiveness in load balancing and failure recovery (§2.5). Notably, datastore systems define shards in an application-transparent manner.

Given the ubiquity of shards, migrating data at shard granularity is an option; in fact, a few systems that do this have been proposed [4, 12, 29, 40]. However, this approach has a serious drawback given typical shard sizes: the vast majority of the migrated data would likely not belong to the working set of the accessing workload at the new location, thus incurring unnecessary migration overhead and wasting inter-datacenter WAN communication bandwidth. At Facebook, because the working set size of accessed data tends to be less than 1MB, migrating an entire shard (1-10GB) would be ineffective.

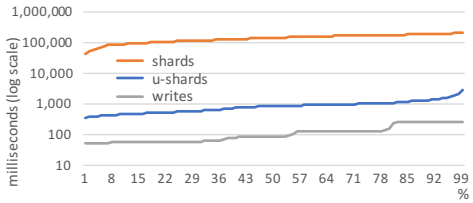


Figure 1: Cumulative distribution of cross-datacenter transfer times. Each curve contains data obtained from 10,000 randomly sampled data points across all cross-datacenter links at Facebook. Avg. shard size is 2GB; avg. µ-shard size is 200KB.

In this paper, by way of Akkio, we advocate for the notion of finer-grained datasets to serve as the unit of migration when managing locality. We call these finer-grained datasets ***µ-shards***. Each µ-shard is defined to contain related data that exhibits some degree of access locality with client applications. It is the application that determines which data is assigned to which µ-shard. At Facebook, µ-shard sizes typically vary from a few hundred bytes to a few megabytes in size, and a µ-shard (typically) contains multiple key-value pairs or database table rows. Each µ-shard is assigned (by Akkio) to a unique shard in that a µ-shard never spans multiple shards.

µ-shards are motivated by our observation that there exist datasets that exhibit good access locality with respect to a client application, but that they are best identified by the client application. Hence, µ-shards are not simply smaller-sized shards. The primary difference between shards and µ-shards, besides size, is the way data is assigned to them. With the former, data is assigned to shards by key partitioning or hashing with little expectation of access locality. With the later, the application assigns data to µ-shards with high expectation of access locality. As a result, µ-shard migration has an overhead that is an order of magnitude lower than that of shard migration (Fig. 1), and its utility is far higher.

µ-shards offer their best advantages in contexts where it is unambiguous how to set the unit of migration so that it is simultaneously as large as possible, meets the constraints of good access locality, and primarily contains data belonging to the same working set of an accessing workload. We have found that there exist many datasets where these parameters are easily identified; see Table 1 for some examples. Because of this, we argue it is propitious to make µ-shards a first class abstraction, such that they are visible to and specified by client applications. The motivation is that only the client applications have the domain knowledge to best determine which data are related and likely to be used together.

Having the application identify related data is not an

[2] Akkio is a play on Harry Potter's Accio Summoning Charm that summons an object to the caster, potentially over a significant distance [31].

[3] In this paper we use the term "underlying datastore system" to refer to the datastore system used natively by the client application. It may be different than the datastore system used by Akkio.

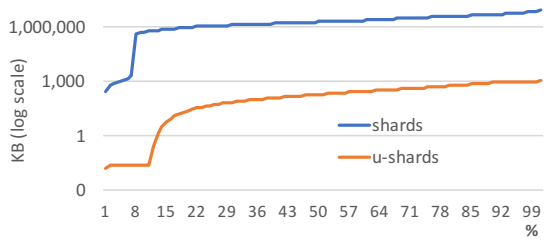


Figure 2: Cumulative distribution of Shard and µ-shard size for ViewState datasets. The ViewState service keeps track of content previously shown to the end-user. ViewState µ-shard sizes tend to be larger than the size of the typical µ-shards managed by Akkio (500KB avg. vs. 200KB avg.).

unreasonable expectation. Many applications already group together data by prefixing keys with a common identifier to ensure that related data are assigned to the same shard. This approach has been used for a long time in practice. Similarly, some databases support the concept of separate partition keys. Spanner supports "directories" although Spanner may shard directories into multiple fragments [11]. Finally, a number of Facebook-internally developed databases, including ZippyDB, support µ-shards as a first class abstraction in the sense that each access request also includes a µ-shard id [3, 8, 34].

Akkio's functionality. Akkio is implemented as a layer between client applications and the underlying datastore system that implements sharding. Although µ-shards are defined by the client applications, Akkio manages them in an application-transparent manner. Akkio is responsible for: (i) tracking client-application accesses to µ-shards so it can take access history into account in its decision making; (ii) deciding where to place each µ-shard; (iii) migrating µ-shards according to a given migration policy for the purpose of reducing access latencies and WAN communication; and (iv) directing each access request to the appropriate µ-shard. Akkio takes capacity constraints and resource loads into account in its placement and migration decisions, even in the face of a heterogeneous environment with a constantly churning hardware fleet.

Akkio is able to support a variety of *replication configurations* and *consistency* requirements (including strong consistency) as specified by each client application service. This flexibility is provided because the client application service owners are in the best position to make the right tradeoffs between availability, consistency, resource cost-effectiveness, and performance. Akkio maps each µ-shard with a specified replication requirement onto a shard configured with the same replication and consistency requirements in the underlying datastore system. As well, it enforces the specified level of consistency during µ-shard migrations.

Other applications. While this paper focuses on

- web application user profile information
- Amazon user browsing history to inform recommendations
- Spotify user listening history to inform subsequent content
- Facebook viewing history to inform subsequent content
- Slack group recent messages
- Reddit subreddits
- email folders
- messaging queues

Table 1: Example datasets conducive to µ-shards. Note that all but the first exhibit relatively low read-write ratios.

Akkio managing locality for geo-distributed environments, Akkio and its mechanisms can be useful in other scenarios. For example, Akkio can be used to migrate µ-shards between cold storage media (e.g. HDDs) and hot storage media (e.g., SSDs) on changes in data temperatures, similar in spirit to CockroachDB's archival partitioning support [38]. Further, for public cloud solutions, Akkio could migrate µ-shards when shifting application workloads from one cloud provider to another cloud provider that is operationally less expensive [39]. Finally, when resharding is required, Akkio could migrate µ-shards, on first access, to newly instantiated shards, allowing a more gentle, incremental form of resharding in situations where many new nodes (e.g. a row of racks) come online simultaneously.

Contributions. We describe the design and implementation of Akkio (§4). To the best of our knowledge, Akkio is the first system capable of managing data locality at µ-shard granularity and at scale, while also supporting strong consistency. In describing Akkio, we focus on scalability; in that sense, this paper focuses on the "plumbing" and not on policy; i.e., specific decision-making algorithms. For applications where Akkio is suitable, we show in §5 that Akkio is:

Effective along a number of dimensions: Compared to typical alternatives, Akkio can achieve *read latency reductions*: up to 50%; *Write latency reductions*: 50% and more; *Cross-datacenter traffic reductions*: by up to 50%. Further, Akkio reduces storage space requirements by up to $X - R$ compared to full replication with X datacenters when a replication factor of R is required for availability.

Scalable: Statistics from production workloads servicing well over a billion users demonstrate the system remains efficient and effective even when processing many tens of millions of requests per second. Akkio can support trillions of µ-shards.

Portable: Akkio's design is simple and flexible enough to allow it to be easily layered on top of most backend datastore systems. Akkio currently runs on top of ZippyDB, Cassandra, and three other internally-developed databases at Facebook.

Limitations. Akkio's approach to managing locality with µ-shards will not be beneficial for all types of data, such as those better served by distributed caches, or

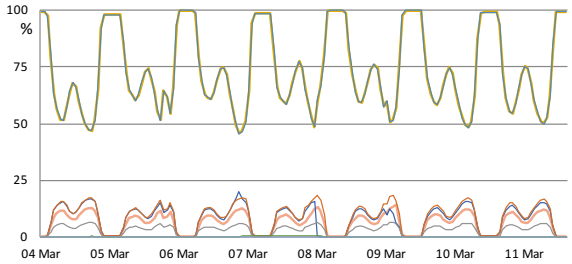


Figure 3: Proportion (in %) of incoming service requests originating from Region A processed at each datacenter. Each curve represents a datacenter. The sum over all curves is always equal to 100%. In this case, Region A has a local datacenter.

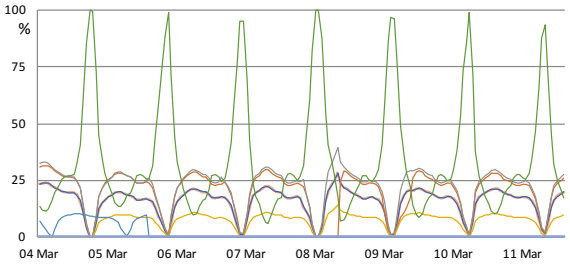


Figure 4: Proportion (in %) of incoming service requests originating from Region B processed at each datacenter. In this case, Region B does not have a local datacenter.

datasets that do not exhibit sufficient access locality. For example, Akkio would not helpful in improving locality for data belonging to the Social Graph. Instead Akkio focuses on workloads with datasets that have low read-write ratios and high access locality. These workloads are quite common and not well served by a caching tier. Further, while Akkio can be layered on top of a variety of datastores, the datastore needs to provide particular features to Akkio as outlined in §4.2. As a result, Akkio may not be able to accommodate all datastore systems. Finally, Akkio does not currently support inter-µ-shard transactions, unless implemented entirely client-side; providing this support is left for future work.

We begin the paper by substantiating our motivation underlying Akkio's approach (§2) and present background needed to understand the rest of the paper (§3).

2 Motivation

2.1 Capital and operational costs matter

Capital and operational costs become consequential when an organization's infrastructure must scale to target a large number of users around the world, justifying considerable efforts to restrain resource usage where possible. Consider an organization with ten datacenters and many hundreds of petabytes of data that must be accessible. While it is difficult to obtain transparent, publicly available pricing information on the true cost of storage, a lower bound for capital depreciation and operational costs could be on the order of two cents per gigabyte per month [9, 28]. This translates to $2 million per 100 petabytes per month. Clearly, replicating all data onto all ten datacenters is difficult to justify from an economic perspective when, in many cases, acceptable durability could be achieved with three replicas.

WAN cross-datacenter links can also be costly and need to be taken into account. For example, estimates for the costs of a 10 Gbps subterranean link vary from $1 to $9 per km per month, depending on route [22]. (To put this into perspective, transferring a 10 GB shard over a 10 Gbps WAN link will consume roughly 10 seconds of bandwidth.) As a result, cross-datacenter link bandwidth will typically be constrained and therefore needs to be used judiciously.

2.2 Service request movements

The datacenters from which data access requests originate can vary over time, even for data accessed on behalf of a unique user. A change in the requesting datacenter can arise, for example, because the user travels from one region to another, or, more likely, because service workload is shifted from a datacenter with high loads to another with lower loads in order to lower service request response latencies. The alternative to shifting workload to other datacenters at peak times would be to increase the capacity of the overloaded datacenter to deal with peak influx of service requests. But this comes with significant operational overheads, which are hard to justify when other datacenters are mostly idle at the same time, given diurnal request patterns.

Figure 3 shows that shifts in traffic occur on a daily basis at Facebook. The figure shows which datacenters processed incoming service requests originating from one particular region over a week. Each curve represents a different datacenter to which the service requests originating from one region were forwarded. The figure shows that during busy periods, as many as 50% of the requests originating from the given region were shifted to remote datacenters (most often located in an adjacent region). The figure also shows that during non-peak times all of the requests are processed by the local datacenter.

Figure 4 shows the same type of information, but for a region with no local datacenter. Because there is no local datacenter, the service requests are distributed to a number of different datacenters. During non-peak times, we see that almost all traffic is serviced from a single, non-local, but nearby datacenter.

We also measured, for each individual end-user, how many datacenters processed service requests issued on behalf of that user over a period of a week (Table 2): over

Num regions:	1	2	3	4
% of users:	46%	42%	10%	2%

Table 2: The percentage of users for which Num regions were contacted to service requests on behalf of the user.

54% of users have their data accessed from two or more regions. Bottom line: there is a reasonable likelihood that requests issued on behalf of one end-user will be processed by multiple distinct datacenters.

2.3 Low read-write ratios

Many important datasets exhibit low read-write ratios (Table 1). As a Facebook-specific example, dataset ViewState (§5.2.1) keeps track of content previously shown to the end-user and has a read-write ratio of 1. Overall, Facebook has on the order of 100PB of periodically accessed data that has a read-write ratio below 5. Note that with low read-write ratios, fully-replicated data would incur significant cross-datacenter communication, as all replicas would have to be updated on writes.

2.4 Ineffectiveness of distributed caches

A common strategy to obtain localized data accesses is to deploy a distributed cache at each datacenter [2, 5, 13, 14, 15, 27, 32]. In practice this alternative is ineffective for most of the workloads important to Facebook. First, unless the cache hit rate in the cache is extremely high, average read latencies will be high if the target data is not located in the local datacenter. Because of this, caching will demand significant hardware infrastructure, as the caches at each datacenter would have to be large enough to hold the working set of the data being accessed from the datacenter.

Second, low read-write ratios lead to excessive communication over cross-datacenter links, because the data being written will, in the common case, be remote.

Finally, many of the datasets accessed by our services require strong consistency. While providing strongly consistent caches is possible, it significantly increases the complexity of the solution, and it incurs a large amount of extra cross-datacenter communication, further exacerbating WAN latency overheads. It is notable that the widely popular distributed caching systems that are scalable, such as Memcached or Redis, do not offer strong consistency. And for good reason.

2.5 Separate locality management layer

Akkio is implemented as a layer between the application service and the underlying distributed datastore system. This raises the question of whether it would make more sense to implement Akkio's functionality directly within the datastore system. Technically, it would be possible, but we argue that this is not a good idea for two reasons.

First, the size of shards are carefully selected by the datastore architects for the purpose of managing load balancing and failure recovery, taking into account the configuration and other metadata needed to manage the shards. Maintaining this data at µ-shard cardinality would come at high storage overheads with 100's of billions of µ-shards vs. 10,000's of shards. Restructuring a datastore system to achieve the level of scale required to support µ-shards across each of its layers would require non-trivial changes.

Second, many application services use data that are not well-served by Akkio-style locality management; e.g., Google search or Facebook's social graph. Hence, it would only make sense to incorporate Akkio's functionality into *specialized* datastore systems; given that datastore system designers optimize for the common case, they would be reluctant to incorporate the additional complexities associated with µ-shards. However, even with a specialized datastore system, legacy issues come into play; in our experiences, application service owners are reluctant to switch away from the underlying datastore system for which their service was tuned and on which they rely for special features or behaviors.

We believe that Akkio bridges the functionality offered by various distributed datastore systems and the application services' desire for (transparent) data locality management to improve response times and reduce WAN datalink overheads.

3 Background

In this section, we briefly review several aspects of shard replication in distributed datastore systems so we can explain Akkio's architecture in §4. In doing so, we introduce some vocabulary we use in subsequent sections. Without loss of generality, we specifically describe how shard replication is handled in ZippyDB, an internally developed scalable key-value store system.[4]

ZippyDB's data is partitioned horizontally, with each partition assigned to a different shard. Each shard may be configured to have multiple replicas, with one designated to be the *primary* and the others referred to as *secondaries*. (See Fig. 5.) We refer to all of the replicas of a shard as a ***shard replica set***, and each replica participates in a shard-specific Paxos group [21, 24, 25]. A write to a shard is directed to its primary replica, which then replicates the write to the secondary replicas, using Paxos to ensure that writes are processed in the same order at each

[4] ZippyDB is used as the database service for hundreds of use cases at Facebook including news products, Instagram services and WhatsApp components. An increasing number of services are being moved onto ZippyDB at Facebook.

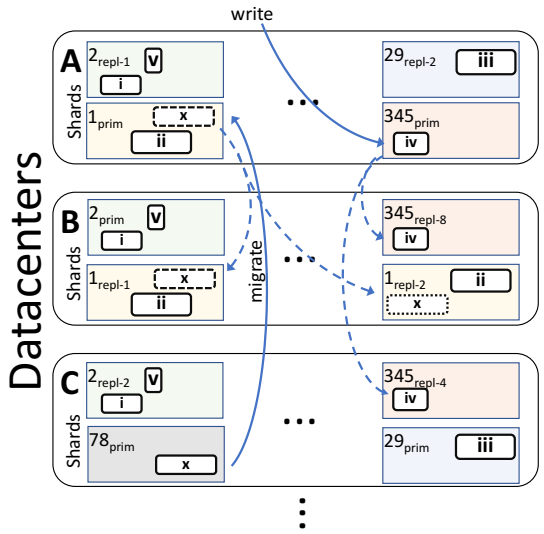


Figure 5: Shards with different replication configurations distributed across datacenters. The shaded rectangles represent shards. Shard 1 has the primary replica in Datacenter A and two secondary replicas in Datacenter B. Shard 2 is replicated across A, B, and C with the primary in B. The smaller boxes represent μ-shards. μ-shard v is assigned to replica set 2; a write that modifies μ-shard v is directed to replica set 2's primary replica and the underlying datastore system replicates the write to the secondary replicas. Akkio is migrating μ-shard x from replica set 78 to replica set 1, and the datastore system replicates x onto 1's secondary.

replica. Reads that need to be strongly consistent are directed to the primary replica. If eventual consistency is acceptable then reads can be directed to a secondary.

A shard's ***replication configuration*** identifies the number of replicas of the shard and how the replicas are distributed over datacenters, clusters, and racks. Shard replication configurations are customizable given that the data owners are in the best position to make the right tradeoffs between availability, consistency, resource-effectiveness, and performance. For example, a service may specify that it requires three replicas, with two replicas (representing a quorum) in one datacenter for improved write latencies and a third in different datacenter for durability. Another service may specify that it requires three replicas located in three different datacenters but that eventual consistency is sufficient. A third might require only one copy, perhaps because the infrastructure overhead of having multiple copies may be deemed to be too high relative to the value of the data.

We use the term ***replica set collection*** to refer to the group of all replica sets that have the same replication configuration. Each such collection is assigned a unique id we call a ***location handle***. When running on top of ZippyDB, Akkio places μ-shards on, and migrates μ-shards between different such replica set collections.

Fig. 5 depicts several shard replica sets and a number of μ-shards within the replica sets. It also shows how a write to a μ-shard is propagated to all secondaries.

Replica sets collections are provisioned and made available to a client application service by a utility that takes input from the client application service owners to help them make the right tradeoffs between availability, consistency, resource-effectiveness, and performance. For example, it inputs application-service parameters that include expected data size, expected access rate (i.e., QPS), R/W-ratios, etc. It also inputs policy parameters that include replication factor, availability requirements, consistency requirements and constraints with respect to where the replicas can be placed.

In general, all possible configurations are included that minimize the replication factor (within the specified constraints). However, some configurations may be excluded. For example, for ViewState, all replica set configurations with three replicas in three different datacenters are excluded so that two replicas will always be located in the same datacenter so that writes have lower latency (given the applications low R/W-ratio).

Once shards have been provisioned, then ZippyDB's *Shard Manager* assigns each shard replica to a specific ZippyDB server while obeying the specified policy rules. The assignment is registered with a *Directory Service* so that the ZippyDB client library embedded in the application service can identify the server to send its access requests to. Shard Manager is also responsible for: (*i*) load balancing, by migrating shards if necessary; and (*ii*) monitoring the liveliness of ZippyDB servers, taking appropriate action when a server failure is detected.

As a final comment, we observe that ZippyDB is able to manage multiple different replication configurations inside a single ZippyDB deployment. Other datastore systems may not be able to support multiple configurations inside a *single* deployment. However, in that case one can usually implement different replication configurations in a straight-forward way by using *multiple* datastore deployments.

4 Akkio Design and Implementation

For clarity, we describe Akkio's design and implementation in the context of a single client application service, ViewState, which uses ZippyDB as its its underlying datastore system. This is without loss of generality, because the underlying database is unaware of Akkio's presence beyond a small portion of code in the database client library.

4.1 Design guidelines

Akkio's design is informed by three primary guidelines. First, Akkio uses an additional level of indirection: it

maps µ-shards onto shard replica set collections whose shards are in turn mapped to datastore storage servers. This allows Akkio to rely on ZippyDB functionality to provide replication, consistency, and intra-cluster load balancing. Secondly, Akkio is structured so as to keep most operations asynchronous and not on any critical path — the only operation in the critical path is the µ-shard location lookup needed for each data access to identify in which replica set collection the target µ-shard is located. Thirdly, Akkio minimizes the intersection with the underlying application datastore tier (e.g., ZippyDB), which makes it more portable. The only two points where the datastore system and Akkio meet are in the datastore client libraries and in Akkio's migration logic which is specific to the datastore.

4.2 Requirements

Akkio imposes three requirements on client application services that wish to use it. First, the client application service must partition data into µ-shards, which are expected to exhibit a fair degree of access locality for Akkio to be effective. Second, the client application service must establish its own µ-shard-id scheme that identifies its µ-shards. µ-shard-ids can be any arbitrary string, but must be globally unique. Finally, to access data in the underlying application database, the client application service must specify the µ-shard the data belongs to in the call to the database client library. For databases that do not support µ-shards natively as ZippyDB does, the function used to access data is modified to include a µ-shard-id as an argument to each access request; e.g., `read(key)` must be modified to `read(µ-shard-id,key)`.

Akkio imposes two requirements on the underlying database. First, the database must ensure µ-shards do not span shards. Because ZippyDB understands the notion of µ-shards, it will never partition µ-shards. Many databases support explicit partition keys that inform the database how to partition data (e.g., MySQL, Cassandra). Yet other databases may recognize key prefixes when partitioning data (e.g., HBase, CockroachDB).

Second, the underlying application database must provide a minimal amount of support so that Akkio can implement migration while maintaining strong consistency. Because the specific features supported by different datastore systems will vary, the µ-shard migration logic that Akkio implements must be specific to the underlying datastore system being supported. For example, some databases, including ZippyDB, offer access control lists (ACLs) and transactions, which are sufficient for implementing µ-shard migration. Other databases, including Cassandra, offer timestamp support for ordering writes, which is also sufficient.

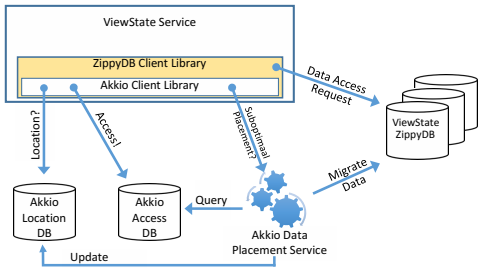


Figure 6: Akkio System Design

4.3 Architectural overview

Akkio's general architecture is depicted in Figure 6. A portion of Akkio's logic is located in the Akkio Client Library, which is embedded into the database client library; i.e., ZippyDB client library, in this case. The client application service makes data access requests by calling the ZippyDB client library, which in turn may make calls to the Akkio Client Library. Beyond the Akkio Client Library, Akkio is made up of three services, which are depicted at the bottom of the figure and described in more detail in the subsections that follow.

The **Akkio Location Service** (ALS) maintains a *location database*. The location database is used on each data access to look up the location of the target µ-shard: the ZippyDB client library makes a call to the Akkio client library `getLocation(µ-shard-id)` function which returns a ZippyDB *location handle* (representing a replica set collection) obtained from the location database. The location handle enables ZippyDB's client library to direct the access request to the appropriate storage server. The location database is updated when a µ-shard is migrated.

An **Access Counter Service** (ACS) maintains an access counter database, which is used to track all accesses so that proper µ-shard placement and migration decisions can be made. Each time the client service accesses a µ-shard, the Akkio client library requests the ACS to record the access, the type of access, and the location from which the access was made. This request is issued asynchronously so that it is not in the critical path.

The ACS is primarily used by Akkio's third service, the **Data Placement Service** (DPS), which decides where to place each µ-shard so as to minimize access latencies and reduce resource usage. The DPS also initiates and manages µ-shard migrations. The Akkio Client Library asynchronously notifies the DPS that a µ-shard placement may be suboptimal whenever a data access request needs to be directed to a remote datacenter. The DPS re-evaluates the placement of a µ-shard only when it receives such a notification. This ensures the DPS triggers migrations only when needed, thus effectively prioritizing migrations and preventing unnecessary migrations for µ-shards that are not being accessed. Note,

however, that a μ-shard access never waits for a potential migration to be evaluated or complete, but proceeds directly with the remote access.

We now discuss these services in more detail.

4.4 Akkio Location Service (ALS)

The Akkio Location Service maintains a database that stores the location handle of each μ-shard. In principle, most any database could be used for storing this information; here we use ZippyDB (without Akkio layered on top of it).[5] The location information is configured to have an eventually consistent replica at every datacenter to ensure low read latencies and high availability, with the primary replicas evenly distributed across all datacenters. This configuration is justified, given the high read-write ratio (> 500) of the database. Moreover, distributed in-memory caches are used at every datacenter to cache the location information so as to reduce the read load on the database, considering that the database needs to be queried on every access request.

It is possible that the distributed cache will serve a stale location mapping, causing the access request to be sent to the wrong server. The target ZippyDB server will determine that the μ-shard is not present from the missing ACL, and will respond accordingly. When that happens, the ZippyDB client library queries the Akkio Location Service again, this time requesting that the cache be bypassed. The client library subsequently re-populates the cache with the latest mapping (making the cache a typical demand-filled look aside cache).

The amount of storage space needed for the ALS is relatively small: each μ-shard requires at most a few hundred bytes of storage, so the size of the dataset for typical client application services will be a few hundred GB. The overhead of maintaining a database for this amount of data in every datacenter is trivial. Similarly, the in-memory caches require no more than a handful of servers per datacenter, since a single machine can service millions of requests per second. The service can easily scale by increasing the number of caching servers.

4.5 Access Counter Service

Access counters are used to keep track of where μ-shards are accessed from and how frequently. To maintain this information, we use the time-windowed counters [7] provided natively by ZippyDB. The counter database uses a separate, dedicated ZippyDB instance, configured to use 3X replication. For each client application service, Akkio stores a single counter per μ-shard per datacenter.

[5] If the application service uses a different underlying datastore system, we use a separate instance of that datastore system for the location database. We do this because the product owners of the underlying datastores were hesitant to allow another system to be in the critical path of data accesses to their system. The two other Akkio services use ZippyDB regardless.

```
consistency_requirements = STRONG;
replication_configurations = {
    "location_handle_a": <A, B, C>
    "location_handle_b": <D, E, F>
    ....
};
access_counter_service = AccessState;
migration_policy = MigrationPolicy(
    microshard_limit=6hours);
```

Listing 1: Akkio Configuration for Sample Service

The amount of storage needed for the counters is on the order of 10's of bytes per μ-shard and datacenter; in our environment less than 200GB per datacenter, which is again trivial. The counter service can easily scale by spreading the counters over a larger number of servers. As an optimization, the number of counters needed and the overhead of incrementing them can be reduced substantially by observing that many of the client application services have identical access patterns. For example, Facebook's AccessState service, which records actions taken in relation to displayed content, has very similar access traffic patterns as ViewState, which records which content was displayed; the traffic of both services is driven by Facebook user traffic. For this reason, Akkio allows a client application service to specify that the counters of another service should be used as a proxy for its own access pattern, in which case the application service does not need a separate set of counters. Moreover, the requests are batched and send-optimized, so the extra communication traffic generated is marginal. (With our workload, ACS adds 0.001% in networking bandwidth.)

4.6 Akkio Data Placement Service (DPS)

Akkio's Data Placement Service is responsible for mapping μ-shards to location handles and for migrating μ-shards in order to improve locality. There is one DPS per Akkio-supported backend datastore system that is shared among all of the application services using instances of that same datastore system. It is implemented as a distributed service with a presence in every datacenter.

The two main interfaces exported by DPS are `createUshard()` and `evaluatePlacement()`. New μ-shards are provisioned on demand when a μ-shard is accessed for the first time; in that case, the Akkio client library receives an `UNKNOWN_ID` response from ALS, so it invokes `createUshard()` (§4.6.1). `EvaluatePlacement()` is invoked by the Akkio client library asynchronously. It first checks whether initiating a migration is permissible, by checking whether the policy allows the target μ-shard to be migrated at that time, and whether the μ-shard is not already in the process of being

migrated. If migration is permissible, it determines the optimal placement for the μ-shard (§4.6.2) and starts the migration (§4.6.3).

DPS stores various information in its datastore system for each μ-shard migration, including locks to prevent multiple concurrent migrations of the same μ-shard, and sufficient information needed to recover the migration should a DPS server fail during the migration (e.g., from and to location handles, lock owners, etc). As well it maintains historical migration data: e.g., time of last migration to limit migration frequency (to allow the prevention of μ-shards ping-ponging).

4.6.1 Provisioning new μ-shards

When a new μ-shard is being created, DPS must decide where to initially place the μ-shard. Our typical strategy is to select a replica set collection with a primary replica local to the requesting client and secondary replica(s) in one of the more lightly loaded datacenters. But, in principle, any available shard replica set collection could be chosen, so using a hash function to distribute initial μ-shard assignments is also a viable strategy.

The primary reason μ-shard provisioning is delegated to DPS is that if any Akkio client library instance were to do this directly, then a race condition might ensue if two or more client instances decide to create the same new μ-shard concurrently. A further advantage of leveraging DPS is that current resource usage can be taken into account when placing the μ-shard.

4.6.2 Determining optimal μ-shard placement

The default policy for selecting a target replica set collection for an existing μ-shard is to choose the one with the highest *score* from among the available replica set collections, excluding those with replicas in datacenters with exceptionally high disk usage or exceptionally high computing loads.[6] Our implementation computes the score in two steps. First, we compute a per-datacenter score by summing the number of times the μ-shard was accessed from that datacenter over the last X days (where X is configurable), weighting more recent accesses more strongly. The per-datacenter scores for the datacenters on which the replica set collection has replicas are then summed to generate a replica set collection score. If there is a clear winner, we pick that winner.

If multiple replica set collections have the same highest score, we take this set of replica set collections and generate, for each, another score using resource usage data. A per-datacenter score is again generated first, which is proportional to the amount of available resources in the datacenter, taking into account, for example, CPU utilization, storage space usage, and IOPS. The per-replica set collection score is then generated by summing the individual datacenter scores on which the replica set collection has a presence. The replica set collection with the highest score is then selected for placing the target μ-shard, or a random one in case of a tie.

[6] Policies can be configured to include specific thresholds that shouldn't be breached; e.g. to not consider datacenters with over n% CPU usage.

```
Atomically:
  a. acquire lock on u-shard
  b. add migration to ongoing migrations list
Set src u-shard ACL to R/O;
Read u-shard from the src
Atomically:
  - write u-shard to dest
  - set dest u-shard ACL to R/O
Update location-DB with new u-shard mapping
Delete source u-shard and ACL
Set destination u-shard ACL to R/W
Atomically:
  a. release lock on u-shard
  b. remove migration from ongoing migr. list
```

Listing 2: μ-shard migration for ZippyDB using ACLs and transactions. Writes are blocked during the migration.

Information on which replica set collections are available is obtained from Configurator [35], a Facebook configuration service that each client application service keeps up to date. Listing 1 shows a simplified Akkio configuration for a sample application service. `Replication_configurations` provides a mapping between location handles and lists of datacenters in which the shard replicas are located. While location handles are opaque to Akkio, it does understand the list of datacenters and uses that information when deciding where to place μ-shards. `Consistency_requirements` specifies that this application service requires strong consistency. `Access_counter_service` specifies which data to use for the access counters. `Migration_policy` specifies a limit on the number of migrations for each μ-shard to once every 6 hours. Migrations may be limited to prevent μ-shard migration ping-ponging.

4.6.3 μ-shard migration

Once the DPS has identified a destination replica set collection for a given μ-shard, it migrates the μ-shard from the source to a destination. Different μ-shard migration methods are used, depending on the functionality of the application service's underlying database. We first describe μ-shard migration for ZippyDB, which offers access control lists (ACL's) and transactions. Other databases are considered further below. In these descriptions, we assume strong consistency of μ-shard data. We also assume the systems run reliably during migration; migration failure handling is described in (§4.6.5).

```
Atomically:
  a. acquire lock on u-shard
  b. add migration to ongoing migrations list
Start double-writing to src & dest
Wait for location info cache TTL to expire
Copy data from source to dest
Switch reading to dest
Wait for location info cache TTL to expire
Switch writing to dest (ending dbl-writes)
Wait for location info cache TTL to expire
Remove src
Atomically:
  a. release lock on u-shard
  b. remove migration from ongoing migr. list
```

Listing 3: μ-shard migration for Cassandra using timestamps and double-writes. Writes are not blocked during the migration. The timestamps are used to merge data when copying

Listing 2 lists the method we first used for ZippyDB. First, a lock is acquired on the μ-shard to prevent other DPS instances from migrating the same μ-shard. (The lock does not prevent the client from reading and writing μ-shard data.) The source μ-shard ACL is then set to read only (R/O). This effectively blocks writes for the duration of the migration; however, the ZippyDB client library embedded in the application will automatically retry the write if the previous attempt was blocked, thus hiding blocked writes from the client application service.[7] The source μ-shard is then read and subsequently written to the destination μ-shard and the destination μ-shard ACL is set to R/O. The location database is updated with the new μ-shard mapping. The source μ-shard and its ACL is deleted, the destination μ-shard ACL is set to R/W, and the migration lock is released.

Not all underlying databases support ACLs. For example, the variant of Cassandra currently used at Facebook does not offer ACLs. Hence, a different migration method is needed. (See Listing 3.) In this case, the migration method takes advantage of the fact that Cassandra offers timestamps natively and can thus allow writes during ongoing migrations. After first acquiring a lock on the μ-shard, the location database information associated with the μ-shard is modified so that client writes are double written to both the source and destination, while reads continue to be directed to the source. The μ-shard data (from before the start of the double-writing) is then copied from the source to the destination. The timestamps associated with each write are used to merge data appropriately. Once the copy is complete, the location database is modified to have reads go to the destination, while continuing double-writing. The location database is modified to have writes only go to the destination. Finally, the data at the source can be deleted at the source, the μ-shard lock can be released, and the migration can be removed from the list of ongoing migrations.

With this method, each time the location database is updated, which occurs three times, it is necessary to wait for the location database TTL to expire to ensure no stale accesses go to the wrong destination. This delay could be avoided if the underlying database supports ACLs (as, e.g., open source Cassandra does), or if cache entries could be reliably invalidated, then the wait times could be reduced substantially. Also note that a potential race condition could occur with double-writes: if a write on the source succeeds, but not on the destination, then the write is observable when reading from the source, but not when later reading from the destination. We address this by always first writing to the destination, before writing to the source, on double writes.

4.6.4 Replica set collection changes

The replica set collections available to the client application service, and in particular the set of replication topologies they represent, will change over time; e.g., to account for shifts in request traffic or because of changes in underlying hardware availability. Adding a new replica set collection is straightforward: it is simply added to the configuration state and the DPS can begin to use it, migrating μ-shards to it when beneficial. Removing a replica set collection is, however, more involved. The replica set collection to be removed is first *disabled* in the configuration, preventing the DPS from selecting this shard replica set collection from future placement decisions. Then, in an off-line process, a DPS `evaluatePlacement()` call is made for each μ-shard in the disabled shard, which will cause the DPS to migrate the μ-shard to another shard replica set collection using the processes described above.

4.6.5 DPS fault recovery

When any of the servers running Akkio's location or counter services ceases to execute (say, due to a hardware or software failure), they can simply be restarted since their data is reliably persisted. The situation is different with a DPS server, since it may have been in the middle of migrating μ-shards.

To deal with this case, every DPS server instance is assigned a monotonically increasing sequence number (which is obtained from a global Zookeeper deployment [19]). This sequence number is persisted with all state related to pending migrations; e.g., in the per μ-shard lock that is acquired prior to beginning of a migration. When a DPS server instance fails, it will be restarted, potentially on a different server, with a higher sequence number. The newly restarted DPS instance will then go through a recovery process where it queries the

[7] With our ViewState workload, which has a very low read-write ratio, writes are retried in 0.007% of all accesses.

Database	Changes to datastore client library	Datastore-specific migration logic
ZippyDB C++	100	1,000
ZippyDB PHP	150	
Cassandra	500	700
Queue datastore	100	250
Datastore-X	100	250

Table 3: Lines of code implementing the two touch points between Akkio and underlying databases.

location database to identify any ongoing migrations that were initiated by the failed DPS server instance but did not complete. The sequence number for any recovered migration is updated in order to avoid any conflicts with a stale, failed DPS server instance.

For each recovered migration, the DPS servers identifies which state to continue the migration on. This is a custom piece of code that is different for each underlying datastore system and migration approach used. For example, in our ACL based approach, the DPS scans the state of the µ-shard in the source backend and the destination backend to identify which steps of the migration had been completed. It then resumes the migration from that point on. In case of errors during a single migration step, we restart the migration. Migrations are typically retried until they succeed (although this is configurable).

5 Evaluation

5.1 Implementation metrics

A benefit of Akkio's design that enhances portability is how lightweight the touchpoints are between Akkio and the underlying databases. Table 3 lists the lines of code (LoC) required for each of the two touchpoints: e.g., the ZippyDB client library only required 100-150 new or modified LoC to accommodate Akkio, and Akkio only required 1,000 or fewer LoC of datastore-specific code for µ-shard migrations in ZippyDB.

5.2 Use cases analysis

We describe the effect Akkio had on 4 different client application services. All of the metrics we present were gathered from our live production systems running at scale, driven by live user traffic. This limits our ability to experiment, so we primarily compare against the systems that were in place before Akkio was introduced.

5.2.1 ViewState

Description: ViewState stores a history of content previously shown to a user. Each time a user is shown some content, an additional snapshot is appended to the ViewState data. The data is used to prioritize subsequent content each time it is displayed to the user. ViewState stores this history, with an average size of 500KB, in ZippyDB.

Requirements: ViewState data is read on the critical path when displaying content, so minimizing read latencies is important. Writes are not on the critical path, but low write latencies are important for the application, as user engagement tends to drops if the content is not "fresh". The data needs to be replicated three ways for durability. Strong consistency is a requirement.

Setup: ViewState uses replica set collections configured with two replicas in one (local) datacenter and a third in a nearby datacenter with the primary preferentially located in the local datacenter. Akkio migrates µ-shards aggressively for ViewState. Having the primary replica be local ensures reads are fast. Having two replicas locally ensures writes are fast given that a quorum exists locally. Having two replicas locally has the further advantage that, should the primary fail, then the other can become primary. In aggregate, 6 different replica set collections are available for Akkio to migrate ViewState µ-shards across when using 6 datacenters.

Having the primary plus a replica in the same datacenter could, however, cause some writes to get lost should an entire datacenter go down: writes that have reached the primary and the other replica in the same datacenter, but have not reached the third replica, will get lost. The ViewState owners were willing to make this tradeoff for this rare scenario.

Result: Originally, ViewState data was fully replicated across six datacenters. Using Akkio with the setup described above led to a 40% smaller storage footprint,[8] a 50% reduction of cross-datacenter traffic, and about a 60% reduction in read and write latencies compared to the original non-Akkio setup. Each remote access notifies the DPS, resulting in approximately 20,000 migrations a second. See Fig. 7. Using Akkio, roughly 5% of the ViewState reads and writes go to a remote datacenter.

5.2.2 AccessState

Description: AccessState stores information with respect to user actions taken in response to content displayed to the user. The information includes the action taken, what content it was related to, a timestamp of when the action was taken, and so on. AccessState data is appended to by a number of different services, but read mostly by the dynamic content display system. AccessState stores the action history, with an average size of 200KB, in ZippyDB. The read-write ratio for AccessState is far lower than it is for ViewState.

Requirements: Reads are on the critical path when deciding what content to display, and hence low read la-

[8] Only 40% because the number of servers couldn't be further reduced due to the CPU becoming the bottleneck.

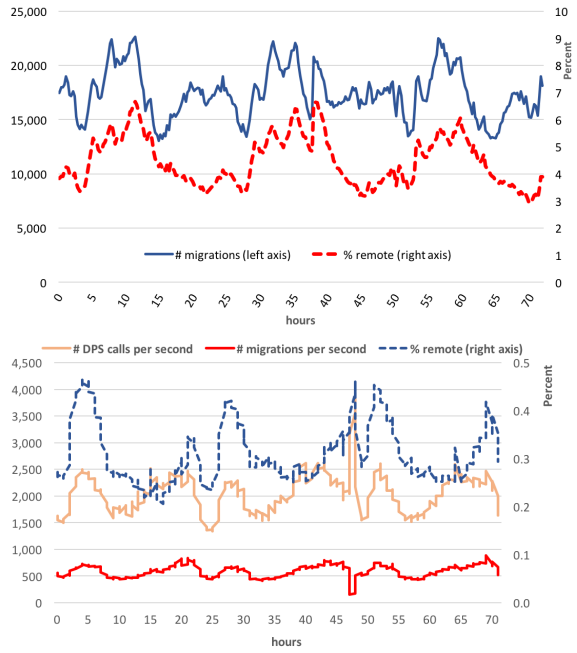


Figure 7: ViewState (top); AccessState (bottom): percentage of accesses to remote data, the number of `evaluatePlacement()` calls to DPS per second, and the number of ensuing µ-shard migrations per second. For ViewState the number of calls to DPS per second and the number of migrations per second are the same.

	avg	p90	p95	p99
With Akkio:	10ms	23ms	26ms	34ms
Without Akkio	76ms	151ms	237ms	371ms

Table 4: AccessState client service access latencies.

tencies are needed. However, writes are not on the critical path and moderate write latency is acceptable (unlike ViewState). The data needs to be replicated three ways but only needs to be eventually consistent.

Setup: AccessState uses replica set collections configured to have three replicas, each one in a different datacenter. Overall, 20 such replica set collections, each with a different topology configuration, plus one replica set collection configured to have a replica in each datacenter, are available for Akkio to migrate AccessState µ-shards. Akkio is configured to not migrate µ-shards aggressively if, based on the access history, it believes the remote processing may be transient. Moreover, it does not migrate the primary replica to the datacenter from which the access was made even though it would lead to lower write latencies, mainly because not doing so significantly reduces the number of migrations needed. (Note that the read-write ratio for AccessState is far higher than it is for ViewState.)

Result: Originally, AccessState data was configured to be fully replicated across six datacenters. Using Akkio with the setup described above led to a 40% decrease in storage footprint, a roughly 50% reduction of cross-datacenter traffic, negligible increase in read latency (0.4%) and a 60% reduction in write latency. Roughly 0.4% of the reads go remote, resulting in about 1,000 migrations a second. Figure 7 shows that there are roughly half as many migrations as there are calls to the DPS.

We also compared AccessState read latencies for a configuration with 3X replication, with and without Akkio. For the configuration without Akkio, the replicas were spread evenly across all datacenters. The results are shown in Table. 4: without Akkio, access latencies are 7X–10X higher.

5.2.3 Instagram Connection-Info

Description: Connection-Info stores data for each user, including when and from where they were online, as well as other status and connection endpoint information. This data is stored on Cassandra. There are roughly 30 billion µ-shards.

Requirements: This application service requires strong consistency, for which it uses Cassandra's quorum read and write features [18]. Intra-continental quorum read and write latencies are important. Originally, this service stored its data using full replication across five datacenters on one continent, but as usage in a second continent increased substantially, some of the data had to be stored on that continent.

Setup: This service uses two replica configurations. One has 5X replication, with a replica in each of five datacenters (as its original setup). The second has 3X replication with two in the second continent and one in the first. Having two replicas together ensures a quorum stays within the same continent in the steady state.

Result: With Akkio it was possible to keep both read and write latencies lower than 50ms which was important to its operation, compared to greater than 100ms which would have been incurred if quorums went across datacenters. This service could not have expanded into the second continent without Akkio.

5.2.4 Instagram Direct

Description: This is a traditional messaging application service that supports group messaging. Each message queue contains the sent messages as well as "cursors" that track the position in the queue for each subscriber. There are roughly 15 billion such queues, but with most queues having a small footprint of a few hundred bytes. The messaging application relies on Iris, a specialized Facebook-internal queuing datastore service that guarantees in-order delivery. (Underneath, Iris uses MySQL for persistent storage.)

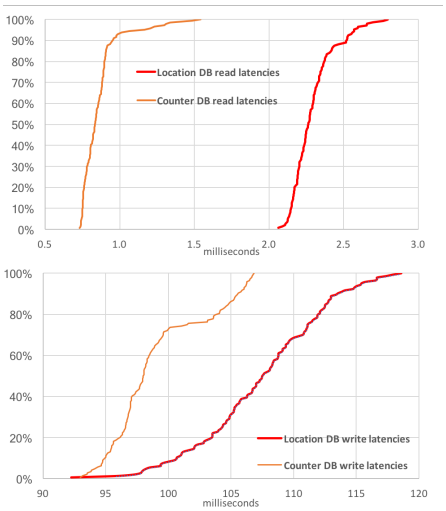


Figure 8: Distribution of client-side latencies for accessing the Akkio location and counter databases, (not taking the cache into account). Read latencies are shown in the top graph; write latencies in the bottom graph.

Requirements: Iris is on the critical path for Instagram Direct end-to-end message delivery. Both low write and low read latencies are thus important. Strong consistency is required.

Setup: Currently, three datacenters are used to store Instagram Direct data. The database is configured to have replica configurations with a primary in each datacenter. Further, each replica set has a secondary replica in the same datacenter as the primary and two additional replica in another datacenter, for a total of four replicas. User access history information is used to decide where to place µ-shards; for message queues that are accessed by multiple users (i.e., group messaging) placement is determined by using each user's access history weighted by rate of user actions.

Result: With Akkio, on average, roughly 3,000 migrations occur per second, resulting in a reduction in end-to-end message delivery latency by 90ms at p90 and 150ms at p95. This, in turn, resulted in user engagement improvements, where the number of message sends increased by 0.9% overall and the number of text message sends increased by 1.1%.

5.3 Analysis of Akkio services

Location Service Using AccessState as an example, the location database uses roughly 200GB storage space (unreplicated) to keep track of the location of each µ-shard, with one µ-shard for each of Facebook's billion+ users. The location database is itself one of the use cases that shares a multi-tenant ZippyDB deployment. It consumes 1,200 fully replicated shards with the primary replicas spread evenly across all regions.

The hit rate of the distributed front-end cache is 98% on average. Read latency on the cache averages to around 1 ms. Figure 8 show the distributions of Akkio client library-side read and write latencies after a miss in the cache. Writes take considerably longer because a quorum needs to be achieved across datacenters before a write is acknowledged.

Step	Time (avg.)
Acquire Lock	151ms
Set Source ACL To Read Only	315ms
Read µ-shard from Source	184ms
Write µ-shard to Destination	130ms
Update Location in DB	151ms
Delete µ-shard From Source	160ms
Set Destination ACL to Read Write	120ms
Release Lock	151ms

Table 5: Breakdown for AccessState µ-shard migration times.

Access Counter Service We present various metrics from the Access Counter DB for AccessState as an example. The amount of storage required for storing one counter for each of the billion+ users and datacenter is about 400GB in total (unreplicated). The Access Counter database also lives in our ZippyDB multi-tenant deployment with 1,100 dedicated shards. Figure 8 depicts the counter database read and write latencies. Neither the reads nor the writes on this database are on any critical path. The read-write ratio is about 1:500. In a typical day, the Access Counter DB for ViewState processes between 300,000 and 550,000 writes per second.

Data Placement Service The DPS receives about 100,000 `evaluatePlacement()` calls per second. However, these calls are asynchronous and not on any critical path. Migrations are the heavy-weight operations executed by the DPS. Table 5 shows the elapsed time breakdown of an AccessState µ-shard migration. The sum of all the individual latencies is relatively high; however, some of the operations can be executed in parallel, different migrations can proceed in parallel, and migration itself is not on the critical path. These latencies have not been an issue for the client application services using Akkio today; optimizing them is left for future work.

6 Related Work

Almost all datastore systems have some form of sharding in order to be scalable, and offer replication to provide high availability; e.g., [6, 10, 16, 30, 33]. However, these systems offer little in terms of locality management. For example, while Cassandra supports fine-grained control of cross-datacenter replication, the control is static and not based on access patterns [23].

A number of systems manage data locality at shard granularity [4, 12, 29, 40]. Given their typical size, we argue that it is challenging to place shards so that most data accesses are local if the number of replicas is limited. Moreover, the overhead of migrating entire shards

is high, and hence these systems tend to be slow to react to shifts in workload.

A few systems manage data locality at a granularity finer than shards. Spanner supports μ-shards in the form of *directories* [11], its unit of data placement. Applications control the contents of a directory using commonality in key prefixes. However, [11] makes no mention of directory-level locality management.

Kadambi et al. extend Yahoo! PNUTs [10] with a per-record selective replication policy [20] but only offer eventual consistency. PNUTs behaves similarly to a distributed cache in that some replicas of records are transient and created on reads and removed when stale; however data resides on disk and a (configurable) minimum number of replicas are kept up to date by propagating updates. The authors argue that collecting and maintaining access statistics of individual records is too complex and incurs too much overhead. Akkio's design shows this need not be. Not tracking these fine grained statistics can lead to sub-optimal decisions.

Volley determines where to place data based on logs that must capture each access [1]. It does this at object granularity. It generates placement and migration recommendations, but leaves the coordination and execution of any resulting migrations to the application, thus making it cumbersome for an application to integrate it. Volley's design to process access logs offline makes it slow to react to shifts in workload and to other real-time events.

Nomad is a prototype distributed key-value store that supports *overlays* as an abstraction [36] designed to hide the protocols needed to coordinate access to data as it is migrated across datacenters. The unit of data management is a *container*, which corresponds to an Akkio μ-shard. However, Nomad does not track access histories or take capacities, loads, and resource-effectiveness into account as Akkio does.

7 Concluding Remarks

This paper makes two key contributions. First, we introduce Akkio, a dynamic locality management service. Second, we introduce and advocate for a finer-grained notion of datasets called μ-shards. To our knowledge, Akkio is the first dynamic data locality system for geo-distributed datastore systems that migrates data at μ-shard granularity, that can offer strong consistency, and that can operate at scale. The system demonstrates that it is possible, and advantageous, to capture data access statistics at fine granularity for making data placement decisions.

Akkio's design is reasonably simple and largely based on techniques well-established in the distributed systems community. Yet we have found it to be effective (§5.2). So far, several hundred application services at Facebook use Akkio, and Akkio manages over 100PB of data. We believe that our choice to implement Akkio as a separate layer between the application services and their underlying databases has worked out well. Separating the concerns of locality management on the one hand, and replication, load-balancing and failure recovery on the other hand, led to a much simpler design and made Akkio viable to a larger set of application services.

With our experiences deploying Akkio, we learned a number of lessons, most of which center around having to make Akkio far more configurable than we had anticipated. (1) we initially planned on storing all of Akkio's metadata in Akkio's own datastore system (ZippyDB). However, we found that application service owners were not willing to add an extra cross-datastore dependency in their critical path (and not willing to change the underlying datastore system they were already using). This forced us to make the location metadata store logic pluggable so that the location metadata could be stored on the application's underlying datastore system. (2) We initially assumed all application services would follow the same migration strategy. However, we found that we had to create a separate migration strategy for each underlying datastore system so as to play to its strengths. (3) We learned that migrations didn't need to be real-time in all cases; e.g., moving messenger conversations to their center of gravity once a day lead to more efficient resource usage, in part because smarter, off-line placement decisions became feasible. More generally, we found that the decision of when to migrate had to be customizable: many application services wanted to delay having their μ-shards migrated by several hundred milliseconds after the first sub-optimal access in order to decrease the chances of the migration interfering with subsequent write accesses (especially if the migration strategy involved taking the μ-shard offline to writes for a small duration). (4) We expected to only need a few different scoring policies when making placement decisions, but ultimately had to support quite a variety of specific scoring policies; e.g., taking recent activity of individual end-users into account when making messaging μ-shard placement decisions. (5) We found that Akkio made capacity planning (growth projections for different datacenters) significantly more difficult with the added dimension of locality, requiring finer-grained estimates of datacenter resource growth.

Going forward, more applications are being moved to run on Akkio, and more datastore systems are being supported (e.g., MySQL). Further, work has started using Akkio (*i*) to migrate data between hot and cold storage, and (*ii*) to migrate data more gracefully onto newly created shards when resharding is required to accommodate (many) new nodes.

Acknowledgements

Many helped contribute to the Akkio system; in particular Victoria Dudin, Harsh Poddar, Dmitry Guyvoronsky; from the ZippyDB team: Sanketh Indarapu, Sumeet Ungratwar, Benjamin Renard, Daniel Pereira, Prateek Jain, Renato Ferreira, Joanna Bujnowska, Igor Pozgaj, Charlie Pisuraj, Tim Mulhern; from the Cassandra team: Dikang Gu, Andrew Whang, Xiangzhou Xia, Abhishek Maloo; from the Generic Iris team: Changle Wang, Jeremy Fein, Kristina Shia; From the Instagram team: Colin Chang, Jingsong Wang; from the Messaging Iris team: Rafal Szymanski, Jeffrey Bahr, Phil Lopreiato, Adrian Wang. We also thank the reviewers, and our shepherd Kang Chen, for their constructive comments that led to a far better paper.

References

[1] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., WOLMAN, A., AND BHOGAN, H. Volley: Automated data placement for geo-distributed cloud services. In *Proc. 7th USENIX Conf. on Networked Systems Design and Implementation (NSDI'10)* (San Jose, California, April 2010), pp. 17–32.

[2] AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. DBProxy: A dynamic data cache for web applications. In *Proc. 19th Intl. Conf. on Data Engineering (ICDE'03)* (Bangalore, India, March 2003), pp. 821–831.

[3] ANNAMALAI, M. ZippyDB: A distributed key-value store. Talk at Data @ Scale: `https://code.facebook.com/posts/371721473024046/inside-data-scale-2015`, June 2015.

[4] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *Proc 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, October 2014), pp. 367–381.

[5] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., ET AL. TAO: Facebook's distributed data store for the social graph. In *Proc. USENIX Annual Technical Conference (USENIXATC'13)* (San Jose, CA, June 2013), pp. 49–60.

[6] CATTELL, R. Scalable SQL and NoSQL data stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[7] CHABCHOUB, Y., AND HEBRAIL, G. Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window. In *Proc. IEEE Intl. Conf. on Data Mining Workshops* (Sydney, Australia, Dec 2010), pp. 1297–1303.

[8] CHEN, G. J., WIENER, J. L., IYER, S., JAISWAL, A., LEI, R., SIMHA, N., WANG, W., WILFONG, K., WILLIAMSON, T., AND YILMAZ, S. Realtime data processing at Facebook. In *Proc. 2016 Intl. Conf. on Management of Data (SIGMOD'16)* (San Francisco, California, 2016), pp. 1087–1098.

[9] CHESTER, D. Considering the real cost of public cloud storage vs. on-premises object storage, June 2017. [Online; posted 23-June-2017].

[10] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proc. of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, Oct 2012), pp. 261–264.

[12] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endowment 3*, 1-2 (Sept. 2010), 48–57.

[13] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symp. on Operating Systems Principles (SOSP'07)* (Stevenson, Washington, 2007), pp. 205–220.

[14] FITZPATRICK, B. Distributed caching with Memcached. *Linux Journal 2004*, 124 (2004), 5.

[15] GARROD, C., MANJHI, A., AILAMAKI, A., MAGGS, B., MOWRY, T., OLSTON, C., AND TOMASIC, A. Scalable query result caching for web applications. *Proc. VLDB Endow.* (Aug. 2008), 550–561.

[16] GEORGE, L. *HBase: The Definitive Guide*, 2nd ed. O'Reilly Media, Inc., 2017.

[17] GOOGLE. Cloud locations. `https://cloud.google.com/about/locations/`. [Online; retrieved 12-April-2018].

[18] HEWITT, E., AND CARPENTER, J. *Cassandra: The Definitive Guide*, 2 ed. O'Reilly Media, 2016.

[19] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX Annual Technical Conference (USENIXATC'10)* (Boston, MA, 2010), pp. 145–158.

[20] KADAMBI, S., CHEN, J., COOPER, B. F., LOMAX, D., RAMAKRISHNAN, R., SILBERSTEIN, A., TAM, E., AND GARCIA-MOLINA, H. Where in the world is my data. In *Proc. 34th Intl. Conf. on Very Large Data Bases (VLDB'11)* (Seattle, Washington, August 2011), pp. 1040–1050.

[21] KIRSCH, J., AND AMIR, Y. Paxos for system builders: An overview. In *Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)* (Yorktown Heights, NY, 2008), ACM, pp. 3:1–3:6.

[22] KREIFELDT, E. Myriad factors conspire to lower submarine bandwidth prices. `http://www.lightwaveonline.com/articles/2016/08/myriad-factors-conspire-to-lower-submarine-bandwidth-prices.html`, August 2016. [Online; posted 31-August-2016 — original source: TeleGeography `https://www.telegeography.com`].

[23] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[24] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[25] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Dec 2001), 51–58.

[26] MARZ, N., AND WARREN, J. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[27] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI,

V. Scaling Memcache at Facebook. In *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation (NSDI'13)* (Lombard, IL, 2013), pp. 385–398.

[28] NUFIRE, T. The cost of cloud storage. `https://www.backblaze.com/blog/cost-of-cloud-storage`, June 2017. [Online; posted 29-June-2017].

[29] P N, S., SIVAKUMAR, A., RAO, S., AND TAWARMALANI, M. D-tunes: Self tuning datastores for geo-distributed interactive applications. In *Proc. of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)* (Hong Kong, 2013), pp. 483–484.

[30] PLUGGE, E., HOWS, D., MEMBREY, P., AND HAWKINS, T. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*, 3rd ed. Apress, 2015.

[31] ROWLING, J. K. *Harry Potter and the Goblet of Fire*. Thorndike Press, 2000.

[32] SHAROV, A., SHRAER, A., MERCHANT, A., AND STOKELY, M. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. of the VLDB Endowment 8*, 12 (2015), 1490–1501.

[33] STRICKLAND, R. *Cassandra 3.x High Availability*, 2nd ed. Packt Publishing Ltd, 2016.

[34] TAI, A., KRYCZKA, A., KANAUJIA, S., PETERSEN, C., ANTONOV, M., WALIJI, M., JAMIESON, K., FREEDMAN, M. J., AND CIDON, A. Live recovery of bit corruptions in datacenter storage systems. *CoRR abs/1805.02790* (2018).

[35] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at Facebook. In *Proc. 25th Symp. on Operating Systems Principles (SOSP'15)* (Monterey, California, 2015), pp. 328–343.

[36] TRAN, N., AGUILERA, M. K., AND BALAKRISHNAN, M. Online migration for geo-distributed storage systems. In *Proc. USENIX Annual Technical Conference (USENICATC'11)* (Portland, Oregon, June 2011), pp. 201–215.

[37] WIKIPEDIA CONTRIBUTORS. Shard (database architecture) — Wikipedia. `https://en.wikipedia.org/w/index.php?title=Shard_(database_architecture)&oldid=845931919`, 2018. [Online; accessed 14-September-2018].

[38] WOODS, A., AND HARRISON, D. How to leverage geo-partitioning. `https://www.cockroachlabs.com/blog/geo-partitioning-two/`, April 2018. [Online; retrieved 12-April-2018].

[39] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. 24th ACM Symp. on Operating Systems Principles (SOSP'13)* (Farminton, Pennsylvania, November 2013), pp. 292–308.

[40] YU, H., AND VAHDAT, A. Minimal replication cost for availability. In *Proc. 21st Annual Symp. on Principles of Distributed Computing (PODC'02)* (Monterey, California, July 2002), pp. 98–107.

Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory

Pengfei Zuo, Yu Hua, Jie Wu
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, China
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Non-volatile memory (NVM) as persistent memory is expected to substitute or complement DRAM in memory hierarchy, due to the strengths of non-volatility, high density, and near-zero standby power. However, due to the requirement of data consistency and hardware limitations of NVM, traditional indexing techniques originally designed for DRAM become inefficient in persistent memory. To efficiently index the data in persistent memory, this paper proposes a write-optimized and high-performance hashing index scheme, called *level hashing*, with low-overhead consistency guarantee and cost-efficient resizing. Level hashing provides a sharing-based two-level hash table, which achieves a constant-scale search/insertion/deletion/update time complexity in the worst case and rarely incurs extra NVM writes. To guarantee the consistency with low overhead, level hashing leverages log-free consistency schemes for insertion, deletion, and resizing operations, and an opportunistic log-free scheme for update operation. To cost-efficiently resize this hash table, level hashing leverages an in-place resizing scheme that only needs to rehash 1/3 of buckets instead of the entire table, thus significantly reducing the number of rehashed buckets and improving the resizing performance. Experimental results demonstrate that level hashing achieves 1.4×−3.0× speedup for insertions, 1.2×−2.1× speedup for updates, and over 4.3× speedup for resizing, while maintaining high search and deletion performance, compared with state-of-the-art hashing schemes.

1 Introduction

As DRAM technology is facing significant challenges in density scaling and power leakage [44, 56], non-volatile memory (NVM) technologies, such as ReRAM [9], PCM [61], STT-RAM [10] and 3D XPoint [1], are promising candidates for building future memory systems. The non-volatility enables data to be persistently stored into NVM as *persistent memory* for instantaneous failure recovery. Due to byte-addressable benefit and the access latency close to DRAM, persistent memory can be directly accessed through the memory bus by using CPU load and store instructions, thus avoiding high overheads of conventional block-based interfaces [18, 39, 63, 64]. However, NVM typically suffers from the limited endurance and low write performance [50, 67].

The significant changes of memory architectures and characteristics result in the inefficiency of indexing data in the conventional manner that overlooks the requirement of data consistency and new NVM device properties [35, 46, 58, 64, 68]. A large amount of existing work has improved tree-based index structures for efficiently adapting to persistent memory, such as CDDS B-tree [58], NV-Tree [64], wB$^+$-Tree [17], FP-Tree [46], WORT [35], and FAST&FAIR [30]. Tree-based index structures are typically with the lookup time complexity of average O(log(N)) where N is the size of data structures [12, 19]. Unlike tree-based index structures, hashing-based index structures are flat data structures, which are able to achieve constant lookup time complexity, i.e., O(1), which is independent of N [42]. Due to providing fast lookup responses, hashing index structures are widely used in main memory systems. For example, they are fundamental components in main memory databases [27, 33, 38, 65], and used to index in-memory key-value stores [7, 8, 25, 36, 66], e.g., Redis and Memcached. However, when hashing index structures are maintained in persistent memory, multiple non-trivial challenges exist which are rarely touched by existing work.

1) High Overhead for Consistency Guarantee. Data structures in persistent memory should avoid any data inconsistency (i.e., data loss or partial updates) when system failures occur [28, 35, 46]. However, the new architecture that NVM is directly accessed through the memory bus causes high overhead to guarantee consistency. First, memory writes are usually reordered by CPU and memory controller [18, 20]. To ensure the

ordering of memory writes for consistency guarantee, we have to employ cache line flush and memory fence, introducing high performance overhead [17, 31, 45, 64]. Second, the atomic write unit for modern processors is generally no larger than the memory bus width (e.g., 8 bytes for 64-bit processors) [17, 20, 24, 60]. If the written data is larger than an atomic write unit, we need to employ expensive logging or copy-on-write (CoW) mechanisms to guarantee consistency [30, 35, 58, 64].

2) Performance Degradation for Reducing Writes. Memory writes in NVM consume the limited endurance and cause higher latency and energy than reads [50, 67]. Moreover, more writes in persistent memory also cause more cache line flushes and memory fences as well as possible logging or CoW operations, significantly decreasing the system performance. Hence, write reduction matters in NVM. Previous work [22, 68] demonstrates that common hashing schemes such as chained hashing, hopscotch hashing [29] and cuckoo hashing [47, 55] usually cause many extra memory writes for dealing with hash collisions. The write-friendly hashing schemes, such as PFHT [22] and path hashing [68], are proposed to reduce NVM writes in hashing index structures but at the cost of decreasing access performance (i.e., the throughput of search, insertion and deletion operations).

3) Cost Inefficiency for Resizing Hash Table. With the increase of the load factor (i.e., the ratio of the number of stored items to that of total storage units) of a hash table, the number of hash collisions increases, resulting in the decrease of access performance as well as insertion failure. Resizing is essential for a hash table to increase the size when its load factor reaches a threshold or an insertion failure occurs [26, 29, 48, 57]. Resizing a hash table needs to create a new hash table whose size is usually doubled, and then iteratively rehash all the items in the old hash table into the new one. Resizing is an expensive operation due to requiring O(N) time complexity to complete where N is the number of items in the hash table. Resizing also incurs N insertion operations, resulting in a large number of NVM writes with cache line flushes and memory fences in persistent memory.

To address these challenges, this paper proposes *level hashing*, a write-optimized and high-performance hashing index scheme with low-overhead consistency guarantee and cost-efficient resizing for persistent memory. Specifically, this paper makes the following contributions:

• ***Low-overhead Consistency Guarantee.*** We propose log-free consistency guarantee schemes for insertion, deletion, and resizing operations in level hashing. The three operations can be atomically executed for consistency guarantee by leveraging the token in each bucket whose size is no larger than an atomic write unit, without the need of expensive logging/CoW. Furthermore, for update operation, we propose an opportunistic log-free scheme to update an item without the need of logging/CoW in most cases. If the bucket storing the item to be updated has an empty slot, an item can be atomically updated without using logging/CoW.

• ***Write-optimized Hash Table Structure.*** We propose a sharing-based two-level hash table structure, in which a search/deletion/update operation only needs to probe at most four buckets to find the target key-value item, and hence has the constant-scale time complexity in the worst case with high performance. An insertion probes at most four buckets to find an empty location in most cases, and in rare cases only moves at most one item, with the constant-scale worst-case time complexity.

• ***Cost-efficient Resizing.*** To improve the resizing performance, we propose a cost-efficient in-place resizing scheme for level hashing, which rehashes only 1/3 of buckets in the hash table instead of the entire hash table, thus significantly reducing NVM writes and improving the resizing performance. Moreover, the in-place resizing scheme enables the resizing process to take place in a single hash table. Hence, search and deletion operations only need to probe one table during the resizing, improving the access performance.

• ***Real Implementation and Evaluation.*** We have implemented level hashing[1] and evaluated it in both real-world DRAM and simulated NVM platforms. Extensive experimental results show that the level hashing speeds up insertions by 1.4×−3.0×, updates by 1.2×−2.1×, and resizing by over 4.3× while maintaining high search and deletion performance, compared with start-of-the-art hashing schemes including BCH [25], PFHT [22] and path hashing [68]. The concurrent level hashing improves the request throughput by 1.6×−2.1×, compared with the start-of-the-art concurrent hashing scheme, i.e., libcuckoo [37].

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 presents the design details. The performance evaluation is shown in Section 4. Section 5 discusses the related work and Section 6 concludes this paper.

2 Background and Motivation

In this section, we present the background of the data consistency issue in persistent memory and hashing index structures.

2.1 Data Consistency in NVM

In order to improve system reliability and efficiently handle possible system failures (e.g., power loss and

[1]The source code of level hashing is available at https://github.com/Pfzuo/Level-Hashing.

system crashes), the non-volatility property of NVM has been well explored and exploited to build persistent memory systems. However, since the persistent systems typically contain volatile storage components, e.g., CPU caches, we have to address the potential problem of data consistency that is interpreted as preventing data from being lost or partially updated in case of a system failure. To achieve data consistency in NVM, it is essential to ensure the ordering of memory writes to NVM [17, 35, 64]. However, the CPU and memory controller may reorder memory writes. We need to use the cache line flush instruction (`CLFLUSH` for short), e.g., *clflush*, *clflushopt* and *clwb*, and memory fence instruction (`MFENCE` for short), e.g., *mfence* and *sfence*, to ensure the ordering of memory writes, like existing state-of-the-art schemes [17, 35, 46, 58, 64]. The `CLFULSH` and `MFENCE` instructions are provided by the Intel x86 architecture [4]. Specifically, `CLFULSH` evicts a dirty cache line from caches and writes it back to NVM. `MFENCE` issues a memory fence, which blocks the memory access instructions after the fence, until those before the fence complete. Since only `MFENCE` can order `CLFLUSH`, `CLFLUSH` is used with `MFENCE` to ensure the ordering of `CLFLUSH` instructions [4]. However, the `CLFLUSH` and `MFEMCE` instructions cause significant system performance overhead [17, 20, 58]. Hence, it is more important to reduce writes in persistent memory.

It is well-recognized that the atomic memory write of NVM is 8 bytes, which is equal to the memory bus width for 64-bit CPUs [17, 35, 46, 58, 64]. If the size of the updated data is larger than 8 bytes and a system failure occurs before completing the update, the data will be corrupted. Existing techniques, such as logging and copy-on-write (CoW), are used to guarantee consistency of the data whose sizes are larger than an atomic-write size. The logging technique first stores the old data (undo logging) or new data (redo logging) into a log and then updates the old data in place. The CoW first creates a new copy of data and then performs updates on the copy. The pointers that point to the old data are finally modified. Nevertheless, logging and CoW have to write twice for each updated data. The ordering of the two-time writes also needs to be ensured using `CLFLUSH` and `MFENCE`, significantly hurting the system performance.

2.2 Hashing Index Structures for NVM

2.2.1 Conventional Hashing Schemes

Hashing index structures are widely used in current main memory databases [23, 27, 33, 38, 65], and key-value stores [7, 8, 25, 36, 51], to provide fast query responses. Hash collisions, i.e., two or more keys are hashed into the same bucket, are practically unavoidable in hashing index structures. *Chained hashing* [32] is a popular scheme to deal with hash collisions, which stores the conflicting items in a linked list via pointers. However, the chained hashing consumes extra memory space due to maintaining the pointers, and decreases access performance when the linked lists are too long.

Open addressing is another kind of hashing scheme to deal with hash collisions without pointers, in which each item has a fixed probe sequence. The item must be in one bucket of its probe sequence. *Bucketized cuckoo hashing (BCH)* [13, 25, 37] is a memory-efficient open-addressing scheme, which has been widely used due to the constant lookup time complexity in the worst case and memory efficiency (i.e., achieving a high load factor). BCH uses f ($f \geq 2$) hash functions to compute f bucket locations for each item. Each bucket includes multiple slots. An inserted item can be stored in any empty slot in its corresponding f buckets. If all slots in the f buckets are occupied, BCH randomly evicts an item in one slot. The evicted item further iteratively evicts other existing items until finding an empty location. For a search operation, BCH probes at most f buckets and hence has a constant search time complexity in the worst case. Due to sufficient flexibility with only two hash functions, $f = 2$ is actually used in BCH [13, 22, 25, 37]. Hence, the BCH in our paper uses two hash functions.

2.2.2 Hashing Schemes for NVM

The mentioned hashing schemes above mainly consider the properties of the traditional memory devices, such as DRAM and SRAM. Unlike them, the new persistent memory systems are tightly related with the significant changes of memory architectures and characteristics, which bring the non-trivial challenges to hashing index structures. For example, NVM typically has limited endurance and incurs higher write latency than DRAM [50, 67]. The chained hashing results in extra NVM writes due to the modifications of pointers and BCH causes *cascading NVM writes* due to frequently evicting and rewriting items for insertion operations, which exacerbate the endurance of NVM and the insertion performance of hash tables [22, 68]. More importantly, the traditional hashing schemes do not consider data consistency and hence cannot directly work on persistent memory.

Hashing schemes [22, 68] have been improved to efficiently adapt to NVM, which mainly focus on reducing NVM writes in hash tables. Debnath et al. [22] propose a *PCM-friendly Hash Table (PFHT)* which is a variant of BCH for reducing writes to PCM. PFHT modifies the BCH to only allow one-time eviction when inserting a new item, which can reduce the NVM writes from frequent evictions but results in low load factor. In order to improve the load factor, PFHT further uses a stash to store the items failing to be inserted into the

Table 1: Comparisons among level hashing and state-of-the-art memory-efficient hashing schemes. *(In this table, "×" indicates a bad performance, "√" indicates a good performance and "−" indicates a moderate performance in the corresponding metrics.)*

	BCH	PFHT	Path hashing	Level hashing
Memory Efficiency	√	√	√	√
Search	√	−	−	√
Deletion	√	−	−	√
Insertion	×	−	−	√
NVM Writes	×	√	√	√
Resizing	×	×	×	√
Consistency	×	×	×	√

hash table. However, PFHT needs to linearly search the stash when failing to find an item in the hash table, thus increasing the search latency. Our previous work [68, 69] proposes the *path hashing* that supports insertion and deletion operations without any extra NVM writes. Path hashing logically organizes the buckets in the hash table as an inverted complete binary tree. Each bucket stores one item. Only the leaf nodes are addressable by hash functions. When hash collisions occur in the leaf node of a path, all non-leaf nodes in the same path are used to store the conflicting key-value items. Thus insertions and deletions in the path hashing only need to probe the nodes within two paths for finding an empty bucket or the target item, without extra writes. However, path hashing offers a low search performance due to the need of traversing two paths until finding the target item for each search operation.

Table 1 shows a high-level comprehensive comparison among these state-of-the-art memory-efficient hashing schemes including BCH, PFHT and path hashing. In summary, BCH is inefficient for insertion due to frequent data evictions. PFHT and path hashing reduce NVM writes in the insertion and deletion operations but at the cost of decreasing access performance. More importantly, these hashing schemes overlook the data consistency issue of hash tables in NVM as well as the efficiency of the resizing operation that often causes a large number of NVM writes. Our paper proposes the level hashing that achieves good performance in terms of all these metrics as shown in Section 3, which is also verified in the performance evaluation as shown in Section 4.

2.2.3 Resizing a Hash Table

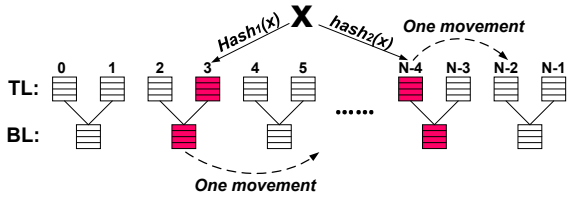
With the increase of the load factor of a hash table, the number of hash collisions increases, resulting in the decrease of the access performance as well as insertion failure [48, 57]. Once a new item fails to be inserted into a hash table, this hash table has to be resized by growing its size. Traditional resizing schemes [40, 48, 53] perform out-of-place resizing, in which expanding a hash table needs to create a new hash table whose size is larger than that of the old one, and then iteratively rehash all items from the old hash table to the new one.

The size of the new hash table is usually double size of the old one [40, 53, 54, 57], due to two main reasons. First, the initial size of a hash table is usually set to be a power of 2, since it allows very cheap modulo operations. For a hash table with power-of-2 (i.e., 2^n) buckets, computing the location of a key based on its hash value, i.e., `hash(key)%2^n`, is a simple bit shift, which is much faster than computing an integral division, e.g., `hash(key)%(2^n-1)`. Thus, if doubling the size in resizing a hash table, the size of the new hash table is still a power of 2. Second, the access performance of a hash table depends on the size of the hash table [57]. If resizing the hash table to a too small size, the new hash table may result in high hash collision rate and poor insertion performance, which will quickly incur another resizing operation. If resizing the hash table to a too large size for inserting a few new items, the new hash table consumes too much memory, reducing the memory space available for other applications. In general, doubling the size when resizing a hash table has been widely recognized [53, 54, 57]. For example, in the real-world applications, such as Java HashMap [5] and Memcached [7], doubling the size is the default setting for resizing a hash table.

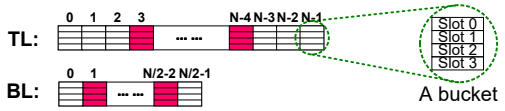
When the stored items are far fewer than the storage units in a hash table, the hash table also needs to be resized via shrinking its size. Resizing is an expensive operation that consumes O(N) time to complete, where N is the number of buckets in the old hash table. Moreover, during the resizing, each search or deletion operation needs to check both old and new hash tables, decreasing the access performance. For hashing index structures maintained in persistent memory, resizing causes a large number of NVM writes with cache line flushes and memory fences, significantly hurting the NVM endurance and decreasing the resizing performance.

3 The Level Hashing Design

We propose *level hashing*, a write-optimized and high-performance hashing index scheme with cost-efficient resizing and low-overhead consistency guarantee for persistent memory. In this section, we first present the basic data structure of level hashing (§3.1), i.e., level hash table, which aims to achieve the high performance as well as high load factor, and rarely incurs extra writes. We then present a cost-efficient in-place resizing scheme (§3.2) for level hashing to reduce NVM writes and improve the resizing performance. We then present the (opportunistic) log-free schemes (§3.3) to reduce the consistency overhead. We finally present the concurrent level hashing leveraging fine-grained locking (§3.4).



(a) The logical structure



(b) The physical storage structure

Figure 1: The hash table structure of level hashing with 4 slots per bucket. *(In these tables, "TL" indicates the top level, and "BL" indicates the bottom level.)*

3.1 Write-optimized Hash Table Structure

A level hash table is a new open-addressing structure with all the strengths of BCH, PFHT and path hashing, including memory-efficient, write-optimized, and high performance, while avoiding their weaknesses, via performing the following major design decisions.

*D_1: **Multiple Slots per Bucket.*** According to multiple key-value workload characteristics published by Facebook [11] and Baidu [34], small key-value items whose sizes are smaller than a cache-line size dominate in current key-value stores. For example, the size of most keys is smaller than 32 bytes, and 16 or 21-byte key with 2-byte value is a common request type in Facebook's key-value store [11]. Motivated by the real-world workload characteristics, we enable the level hash table to be cache-efficient by setting multiple slots in each bucket, e.g., 4 slots per bucket as shown in Figure 1. Thus a bucket can store multiple key-value items each in one slot. When accessing a bucket in the level hash table, multiple key-value items in the same bucket can be prefetched into CPU caches in one memory access, which improves the cache efficiency and thus reduces the number of memory accesses.

*D_2: **Two Hash Locations for Each Key.*** Since each bucket has k slots, the hash table can deal with at most $k-1$ hash collisions occurring in a single hash position. It is possible that more than k key-value items are hashed into the same position. In this case, insertion failure easily occurs, resulting in a low load factor. To address this problem, we enable each key to have two hash locations via using two different hash functions, i.e., $\texttt{hash}_1()$ and $\texttt{hash}_2()$, like BCH [13, 25, 37], PCHT [22] and path hashing [68, 69]. A new key-value item is inserted into the less-loaded bucket between the two hash locations [14]. Due to the randomization of two independent hash functions, the load factor of hash table is significantly improved as shown in Figure 2.

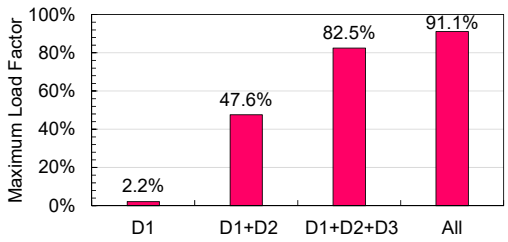


Figure 2: The maximum load factors when adding different design decisions. *(D_1: a one-level hash table with 4 slots per bucket; D_1+D_2: a hash table with design decisions D_1 and D_2; $D_1+D_2+D_3$: a hash table with D_1, D_2 and D_3; All: level hash table that uses $D_1+D_2+D_3+D_4$.)*

*D_3: **Sharing-based Two-level Structure.*** The buckets in the level hash table are divided into two levels, i,e., a top level and a bottom level, as shown in Figure 1a. Only the buckets in the top level are addressable by hash functions. The bottom level is not addressable and used to provide standby positions for the top level to store conflicting key-value items. Each bottom-level bucket is shared by two top-level buckets, and thus the size of the bottom level is half of the top level. If a hash collision occurs in a top-level bucket and all positions in the bucket are occupied, the conflicting key-value item can be stored in its corresponding standby bucket in the bottom level. By using the two-level structure, the load factor of hash table is significantly improved as shown in Figure 2. Moreover, since each addressable bucket has one standby bucket, a search operation only needs to probe at most four buckets, having the constant-scale time complexity in the worst case.

*D_4: **At Most One Movement for Each Successful Insertion.*** To enable key-value items to be evenly distributed among buckets, if both buckets are full during inserting an item in the BCH [13, 22, 25, 37], BCH iteratively evicts one of existing items and thus incurs cascading writes, which is not friendly for NVMs. To avoid the problem of the cascading writes, instead, level hashing allows the movement of at most one item for each insertion. Specifically, during inserting a new item (I_{new}), if the two top-level buckets are full, we check whether it is possible to move any key-value item from one of its two top-level buckets to its alternative top-level location. If no movement is possible, we further insert the new item I_{new} into the bottom level. If the two bottom-level buckets are full, we also check whether it is possible to move any key-value item from one of its two bottom-level buckets to its alternative bottom-level location. If the movement still fails, the insertion fails and the hash table needs to be resized. Note that the movement is saved if the alternative location of the moved item has no empty slot. Allowing one movement

redistributes the items among buckets, thus improving the maximum load factor, as shown in Figure 2.

Put them all together, the hash table structure of level hashing is shown in Figure 1. Figure 1a shows the logical structure of a level hash table that contains two-level buckets. The links between two levels indicate the bucket sharing relationships, instead of pointers. Figure 1b shows the physical storage of a level hash table, in which each level is stored in a one-dimensional array. For a key-value item with the key K, its corresponding two buckets in the top level (i.e., the No.L_{t1} and No.L_{t2} buckets) and its two standby buckets in the bottom level (i.e., the No.L_{b1} and No.L_{b2} buckets) can be obtained via the following equations:

$$L_{t1} = \texttt{hash}_1\texttt{(K)\%N}, L_{t2} = \texttt{hash}_2\texttt{(K)\%N} \quad (1)$$

$$L_{b1} = \texttt{hash}_1\texttt{(K)\%(N/2)}, L_{b2} = \texttt{hash}_2\texttt{(K)\%(N/2)} \quad (2)$$

The computations of Equations 1 and 2 only require the simple bit shift operation since N is a power of 2. The simple yet efficient hash table structure shown in Figure 1 has the following strengths:

• *Write-optimized.* Level hashing does not cause the cascading writes via allowing at most one movement for each insertion. Moreover, only a very small number of insertions incur one movement. Based on our experiments, when continuously inserting key-value items into a level hash table until reaching its maximum load factor, only 1.2% of insertions incur one movement.

• *High-performance.* For a search/deletion/update operation, level hashing probes at most four buckets to find the target item. For an insertion operation, level hashing probes at most four buckets to find an empty location in most cases, and in rare cases further moves at most one existing item. Hence, level hashing achieves the constant-scale worst-case time complexity for all operations.

• *Memory-efficient.* In the level hash table, two hash locations for each key enables the key-value items in the top level to be evenly distributed [43]. Each un-addressable bucket is shared by two addressable buckets to store the conflicting items, which enables the items in the bottom level to be evenly distributed. Allowing one movement enables items to be evenly redistributed. These design decisions enable the level hash table to be load-balanced and memory-efficient, thus achieving more than 90% load factor as shown in Figure 2.

Moreover, the level hashing has a good resizing performance via a cost-efficient in-place resizing scheme as shown in Section 3.2. We guarantees the data consistency in the level hashing with low overhead via the (opportunistic) log-free schemes as shown in Section 3.3.

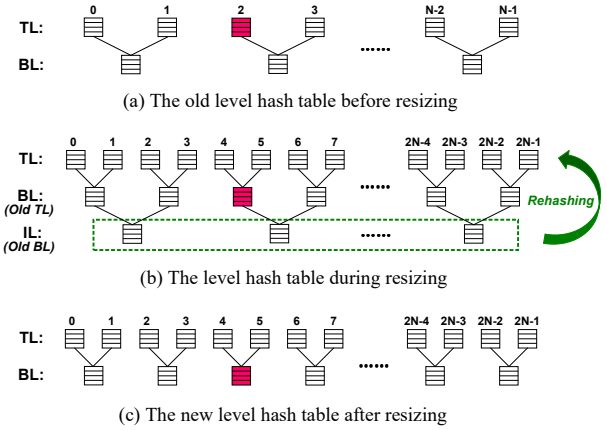


Figure 3: The cost-efficient in-place resizing in the level hashing. *("IL" indicates the interim level.)*

3.2 Cost-efficient In-place Resizing

To reduce NVM writes and improve the resizing performance, we propose *a cost-efficient in-place resizing scheme*. The basic idea of the in-place resizing scheme is to put a new level on the top of the old hash table and only rehash the items in the bottom level of the old hash table when expanding a level hash table.

1) An Overview of Resizing. A high-level overview of the in-place resizing process in the level hashing is shown in Figure 3. Before the resizing, the level hash table is a two-level structure, including a top level (TL) with N buckets and a bottom level (BL) with $N/2$ buckets, as shown in Figure 3a. During the resizing, we first allocate the memory space with $2N$ buckets as the new top level and put it on the top of the old hash table. The level hash table becomes a three-level structure during the resizing, as shown in Figure 3b. The third level is called the interim level (IL). The in-place resizing scheme rehashes the items in the IL into the top-two levels. Each rehashing operation includes reading an item in the IL, inserting the item into the top-two levels and deleting the item from the IL. After all items in the IL are rehashed into the top-two levels, the memory space of the IL is reclaimed. After the resizing, the new hash table becomes a two-level structure again, as shown in Figure 3c. The rehashing failure (which indicates a rehashed item fails to be inserted into the top-two levels) does not occur when the resizing is underway, since currently the total number of stored items is smaller than half of the total size of the new level hash table, and level hashing is able to achieve the load factor of higher than 0.9 (> 0.5) as evaluated in Section 4.2.1.

We observe that the new hash table with $3N$ buckets is exactly double size of the old hash table with $1.5N$ buckets, which meets the demand of real-world applications as discussed in Section 2.2.3. Unlike the

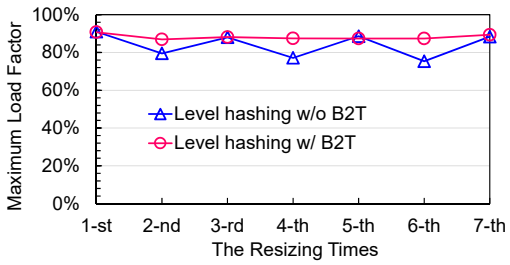


Figure 4: The load factors when the resizings occur.

traditional out-of-place resizing scheme [48] in which the resizing occurs between the old and new tables, the in-place resizing enables the whole resizing process to occur in a single hash table. Thus during resizing, search and deletion operations only need to probe one table and compute the hash functions once, thus improving the access performance. Moreover, the in-place resizing rehashes only the bottom level of the old hash table instead of the entire table. The bottom level only contains $1/3(=0.5N/1.5N)$ of all buckets in the old hash table, thus significantly reducing data movements and NVM writes during the resizing, as well as improving the resizing performance.

We can also shrink the level hash table in place which is an inverse process of expanding the level hash table. Specifically, to shrink the level hash table, we first allocate the memory space with $N/4$ buckets as the new bottom level which is placed on the bottom of the old hash table. We then rehash all items in the old top level into the bottom-two levels.

2) Improving the Maximum Load Factor after Resizing. In the level hash table, each item is stored in the bottom level only when its corresponding two top-level buckets are full. Thus before resizing, the top-level buckets are mostly full and the bottom-level buckets are mostly non-full. After resizing, the top level in the old hash table becomes the bottom level in the new hash table as shown in Figure 3. Thus the bottom-level buckets in the new hash table are mostly full, which easily incur an insertion failure, reducing the maximum load factor. The blue line in Figure 4 shows the load factors of the level hash table when the multiple successive resizings occur. We observe that the maximum load factors in the 2-nd, 4-th, and 6-th resizings are reduced, compared with those in the 1-st, 3-rd and 5-th resizings. The reason is that the bottom-level buckets are mostly full in the 2-nd, 4-th and 6-th resizings.

To address this problem, we propose *a bottom-to-top movement (B2T) scheme* for level hashing. Specifically, during inserting an item, if its corresponding two top-level buckets (L_{t1} and L_{t2}) and two bottom-level buckets (L_{b1} and L_{b2}) are full, the B2T scheme tries to move one existing item (I_{ext}) in the bottom-level bucket L_{b1} or L_{b2} into the top-level alternative locations of I_{ext}. Only when the corresponding two top-level buckets of I_{ext} have no empty slot, the insertion is considered as a failure and incurs a hash table resizing. By performing the B2T scheme, the items between top and bottom levels are redistributed, thus improving the maximum load factor. The red line in Figure 4 shows the load factors when the resizings occur via using the B2T scheme. We observe that the maximum load factors in the 2-nd, 4-th and 6-th resizings are improved.

3) Improving the Search Performance after Resizing. After resizing, the search performance possibly decreases. This is because in the original search scheme (called *static search*) as shown in Section 3.1, we always first probe the top level, and if not finding the target item, we then probe the bottom level. Before resizing, about $2/3$ items are in the top level. However, the $2/3$ items are in the bottom level after resizing, since the top level in the old hash table becomes the bottom level in the new one as shown in Figure 3. Hence, a single search needs to probe two levels in most cases (i.e., about $2/3$ probability) after resizing, thus degrading the search performance.

To address this problem, we propose *a dynamic search scheme* for level hashing. Specifically, for a search, we study two cases based on the numbers of items in the top and bottom levels. First, if the items in the bottom level are more than those in the top level, we first probe the bottom level (based on Equation 2), and if not finding the target item, we then probe the top level (based on Equation 1). Second, if the items in the bottom level are less than those in the top level, we first probe the top level and then the bottom level. Thus after resizing, the items in the bottom level are more than those in the top level and hence we first probe the bottom level, thus improving the search performance. We also demonstrate the performance improvement in Section 4.2.4.

3.3 Low-overhead Consistency Guarantee

In the open-addressing hash tables, a token associated with each slot is used to indicate whether the slot is empty [25, 68]. As shown in Figure 5, in a bucket, the header area stores the tokens of all slots and the remaining area stores the slots each with a key-value item. A token is defined as a 1-bit flag that indicates whether the corresponding slot is empty. For example, the token '0' indicates the corresponding slot is empty and the token '1' indicates the slot is non-empty. The header area is 1 byte when the number of slots is not larger than 8, and 2 bytes for the buckets with 16 slots. Since the header area is always smaller than 8 bytes, modifying the tokens only needs to perform an atomic write. But the key-value items are usually larger than 8 bytes. A straightforward approach is to guarantee the consistency of writing key-value items via logging

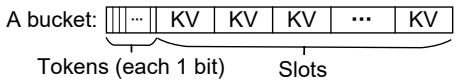


Figure 5: The storage structure of each bucket.

or CoW, which however incurs significant performance overhead as discussed in Section 2.1.

To reduce the overhead of guaranteeing consistency in level hashing, we propose *log-free consistency guarantee schemes* for deletion, insertion, and resizing operations, and *an opportunistic log-free guarantee scheme* for update operation, by leveraging the tokens to be performed in the atomic-write manner.

1) Log-free Deletion. When deleting a key-value item from a slot, we change the token of the slot from '1' to '0', which invalidates the deleted key-value item. The deletion operation only needs to perform an atomic write to change the token. After the token of the slot is changed to '0', the slot becomes available and can be used to insert a new item.

2) Log-free Insertion. There are two cases when inserting a new item into the level hash table.

a) No item movement: The insertion incurs no movement, i.e., inserting a new item to an empty slot. In this case, we first write the new item into the slot and then change its token from '0' to '1'. The ordering of writing the item and changing the token is ensured via an `MFENCE`. Although the new item is larger than 8 bytes, writing the item does not require logging or CoW, since the item becomes valid until the token is set to '1'. If a system failure occurs during writing the item, this item may be partially written but invalid since the current token is '0' and this slot is still available. Hence, the hash table is in a consistent state when system failures occur.

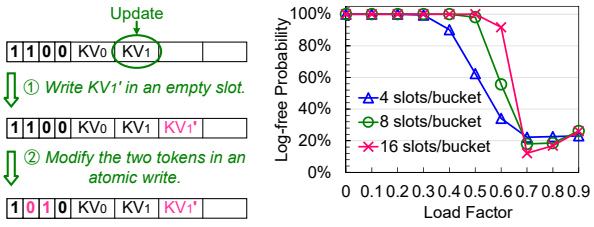
b) Moving one item: The insertion incurs the movement of one item. In this case, we need to take two steps to insert an item, and the ordering of executing the two steps is ensured via an `MFENCE`. The first step is to move an existing item into its alternative bucket. We use $\texttt{slot}_{\texttt{cur}}$ to indicate the current slot of the existing item and use $\texttt{slot}_{\texttt{alt}}$ to indicate its new slot in the alternative bucket. Moving this item first copies the item into $\texttt{slot}_{\texttt{alt}}$, then modifies the token of $\texttt{slot}_{\texttt{alt}}$ from '0' to '1' and finally modifies the token of $\texttt{slot}_{\texttt{cur}}$ from '1' to '0'. If a system failure occurs after changing the token of $\texttt{slot}_{\texttt{alt}}$ before changing the token of $\texttt{slot}_{\texttt{cur}}$, the hash table contains two duplicate key-value items, which however does not impact on the data consistency. It is because when searching this key-value item, the returned value is always correct whichever one of the two items is queried. When updating this item, one of the two items is first deleted and the other one is then updated, as presented in Section 3.3(4). After moving this existing item, the second step inserts the new item into the empty slot using the method of "*a) no item movement*".

3) Log-free Resizing. During resizing, we need to rehash all key-value items in the interim level. For a rehashed item, we use $\texttt{slot}_{\texttt{old}}$ to indicate its old slot in the interim level and use $\texttt{slot}_{\texttt{new}}$ to indicate its new slot in the top-two levels. Rehashing an item in the interim level can be decomposed into two steps, i.e., inserting the item into $\texttt{slot}_{\texttt{new}}$ (*Log-free Insertion*) and then deleting the item from $\texttt{slot}_{\texttt{old}}$ (*Log-free Deletion*). To guarantee the data consistency during a rehashing operation, we first copy the key-value item of $\texttt{slot}_{\texttt{old}}$ into $\texttt{slot}_{\texttt{new}}$, and then modifies the token of $\texttt{slot}_{\texttt{new}}$ from '0' to '1' and finally modifies the token of $\texttt{slot}_{\texttt{old}}$ from '1' to '0'. The ordering of the three steps is ensured via `MFENCE`s. If a system failure occurs when copying the item, the hash table is in a consistent state since the $\texttt{slot}_{\texttt{new}}$ is still available and the item in $\texttt{slot}_{\texttt{old}}$ is not deleted. If a system failure occurs after changing the token of $\texttt{slot}_{\texttt{new}}$ before changing the token of $\texttt{slot}_{\texttt{old}}$, $\texttt{slot}_{\texttt{new}}$ is inserted successfully but the item in $\texttt{slot}_{\texttt{old}}$ is not deleted. There are two duplicate items in the hash table, which however has no impact on the data consistency, since we can easily remove one of the two duplicates after the system is recovered without scanning the whole hash table. In case of a system failure, only the first item (I_{first}) to be rehashed in the interim level may be inconsistent. To check whether there are two duplicates of I_{first} in the hash table, we only need to query the key of I_{first} in the top-two levels. If two duplicates exist, we directly delete I_{first}. Otherwise, we rehash it. Therefore, the hash table can be recovered in a consistent state.

4) Opportunistic Log-free Update. When updating an existing key-value item, if the updated item has two copies in the hash table, we first delete one and then update the other. If we directly update the key-value item in place, the hash table may be left in the corrupted state when a system failure occurs, since the old item is overwritten and lost, and the new item is not written completely. Intuitively, we address this problem via first writing the new or old item into a log and then updating the old item in place, which however incur high performance overhead.

To reduce the overhead, we leverage *an opportunistic log-free update scheme* to guarantee consistency. Specifically, for an update operation (e.g., updating KV_1 to KV_1'), we first check whether there is an empty slot in the bucket storing the old item (KV_1).

- **Yes.** If an empty slot exists in the bucket as shown in Figure 6a, we directly write the new item (KV_1') into the empty slot, and then modify the tokens of the old item (KV_1) and new item (KV_1')



(a) Log-free update (b) The log-free probability

Figure 6: The opportunistic log-free update scheme. *((a) The log-free update scheme; (b) The probability of performing log-free update with the increase of load factor and the change of the number of slots/bucket.)*

simultaneously. The two tokens are stored together and hence can be simultaneously modified in an atomic write. The ordering of writing the new item and modifying the tokens is ensured by an `MFENCE`.

- **No.** If no empty bucket exists in the bucket storing the old item (KV_1), we first log the old item and then update the old item in place. If a system failure occurs during overwriting the old item, the old item can be recovered based on the log.

In summary, if there is an empty slot in the bucket storing the item to be updated, we update the item without logging. We evaluate the opportunity to perform log-free update, i.e., the probability that the bucket storing the updated item contains at least one empty slot, as shown in Figure 6b. The probability is related with the number of slots in each bucket and the load factor of hash table. We observe that when the load factor of hash table is smaller than about $2/3$, the probability of log-free update is very high and decreases with the increase of the load factor and the decrease of the number of slots in each bucket. However, when the load factor is larger than $2/3$, the probability increases with the increase of the load factor. This is because the number of storage units in the top level is $2/3$ of the total storage units. When the load factor is beyond $2/3$, more items are inserted into the bottom level, and the buckets in the bottom level have the higher probability to contain an empty slot than those in the top level.

We further discuss whether the proposed consistency-guarantee schemes work on other hashing schemes. 1) The proposed log-free deletion scheme can be used in other open-addressing hashing schemes, since deletion only operates on a single item. 2) The opportunistic log-free update scheme can be used in other multiple-slot hashing schemes, e.g, BCH, and PFHT. 3) Obviously, the log-free insertion scheme can be used in the hashing schemes without data evictions during insertions, e.g., path hashing, and the hashing schemes with at most one eviction, e.g., PFHT. In fact, the log-free insertion

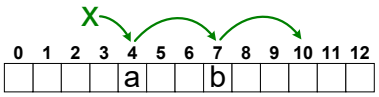


Figure 7: An insertion in the cuckoo hashing.

scheme can also be used in the hashing schemes with iterative eviction operations during insertions, e.g., cuckoo hashing. Specifically, an insertion in cuckoo hashing may iteratively evict key-value items until finding an empty location. The sequence of evicted items is called a cuckoo path [37]. To perform log-free insertion, we first search for a cuckoo path with an empty location but do not execute evictions during search. We then perform evictions starting with the last item in the cuckoo path and working backward toward the first item. For example, as shown in Figure 7, the new item x is inserted into the location $L4$, and the sequence of $x \rightarrow a \rightarrow b \rightarrow \varnothing$ is a cuckoo path. To perform log-free insertion, we first move b from $L7$ to $L10$, and then move a from $L4$ to $L7$, and finally insert x into $L4$.

3.4 Concurrent Level Hashing

As current systems are being scaled to larger number of cores and threads, concurrent data structures become increasingly important [15, 25, 37, 41]. The level hash table does not use pointers and has no cascading writes, which enables level hashing to efficiently support multi-reader and multi-writer concurrency via simply using fine-grained locking.

In the concurrent level hashing, the conflicts occur when different threads concurrently read/write the same slot. Hence, we allocate a fine-grained locking for each slot. When reading/writing a slot, the thread first locks it. Since level hashing allows each insertion to move at most one existing item, an insertion operation locks at most two slots, i.e., the current slot and the target slot that the item will be moved into. Nevertheless, the probability that an insertion incurs a movement is very low as presented in Section 3.1. An insertion locks only one slot in the most cases, and hence the concurrent level hashing delivers high performance as evaluated in Section 4.2.7.

4 Performance Evaluation

4.1 Experimental Setup

All our experiments are performed on a Linux server (kernel version 3.10.0) that has four 6-core Intel Xeon E5-2620 2.0GHz CPUs (each core with 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache), 15MB last level cache and 32GB DRAM.

Since the real NVM device is not available for us yet, we conduct our experiments using Hewlett Packard's Quartz [2, 59], which is a DRAM-based performance

emulator for persistent memory and has been widely used [31, 35, 39, 52, 60]. Quartz emulates the latency of persistent memory by injecting software created delays per epoch and limiting the DRAM bandwidth by leveraging DRAM thermal control registers. However, the current implementation of Quartz [2] does not yet support the emulation of write latency in the persistent memory. We hence emulate the write latency by adding an extra delay after each `CLFLUSH` instruction, following the methods in existing work [31, 35, 39, 52, 60].

The evaluation results in PFHT [22] and path hashing [68] demonstrated that PFHT and path hashing significantly outperform other existing hashing schemes, including chained hashing, linear probing [49], hopscotch hashing [29] and cuckoo hashing [47, 55], in NVM. Therefore, we compare our proposed level hashing with the state-of-the-art NVM-friendly schemes, i.e., PFHT and path hashing, and the memory-efficient hashing scheme for DRAM, i.e., BCH, in both DRAM and NVM platforms. Since these hashing schemes do not consider the data consistency issue on persistent memory, we implement persistent BCH, PFHT, and path hashing using our proposed consistency guarantee schemes as discussed in Section 3.3 for fairly comparing their performance on persistent memory. Moreover, we also compare the performance of these hashing schemes without crash consistency guarantee in DRAM.

Since 16-byte key has been widely used in current key-value stores [11, 34, 62], we use the 16-byte key, the value that is no longer than 15 bytes, and 1-bit token for each slot. Two slots align a cache line (64B) via padding several unused bytes. Every hash table is sized for 100 million key-value items and thus needs about 3.2GB memory space. Besides examining the single-thread performance of each kind of operation, we also use YCSB [21], a benchmark for key-value stores, to evaluate the concurrent performance of the concurrent level hashing in multiple mixed workloads. In the experimental results, each data value is the average of 10-run results.

4.2 Experimental Results

4.2.1 Maximum Load Factor

The maximum load factor is an important metric for hash table due to directly affecting the number of key-value items that a hash table can store and the hardware cost [25, 37]. For evaluating the maximum load factor, we insert unique string keys into empty BCH, PFHT, level and path hash tables until an insertion failure occurs. Specifically, BCH reaches the maximum load factor when a single insertion operation fails to find an empty slot after 500 evictions [25, 37]. For PFHT, the 3% space of the total hash table size is used as a stash,

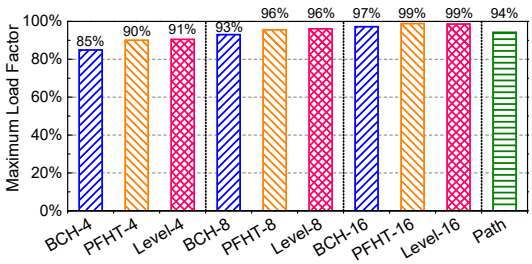


Figure 8: Maximum load factors of hash tables. *(# in the NAME-# indicates the number of slots per bucket.)*

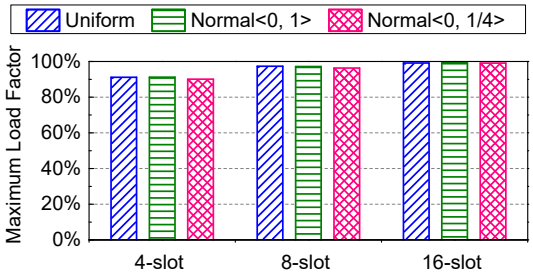
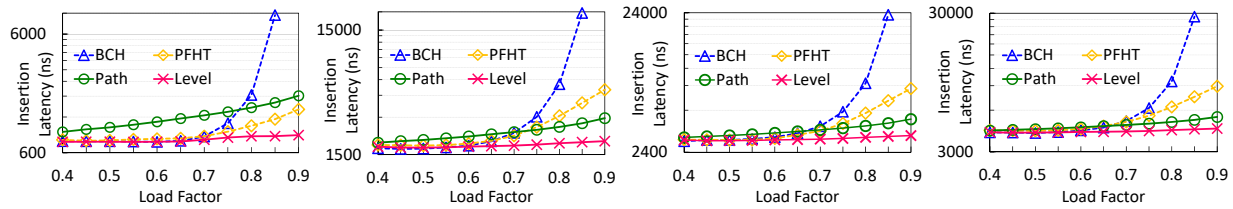


Figure 9: Maximum load factors of the level hash table with different-distribution integer keys. *(Normal$< x, y >$ indicates the logarithmic normal distribution with the parameters $\mu = x$ and $\sigma = y$.)*

following the configuration in the original paper [22]. PFHT reaches the maximum load factor when the stash is full. Level and path hash tables reach the maximum load factors when a single insertion fails to find an empty slot or bucket.

Figure 8 shows that all the four hash tables can achieve over 90% of maximum load factor. Figure 8 also compares different hash tables with the different numbers of slots in each bucket. More slots in each bucket incur higher maximum load factor for BCH, PFHT and level hash table. For the same number of slots in each bucket, PFHT and level hash table have approximately the same maximum load factor, which are higher than BCH. Path hash table is a one-item-per-bucket table and achieves up to 94.2% maximum load factor.

We also evaluate the maximum load factors of the level hash table with different-distribution integer keys including uniform and skewed normal key distributions, as shown in Figure 9. We observe that the level hash table achieves the approximate maximum load factors for the different key distributions. The reason is that hash functions map keys to random hash values, and hence whatever the key distribution is, the generated hash value distribution is still randomized. Keys are then randomly distributed among buckets of hash table based on their hash values. Therefore, the skewed key distribution doesn't result in the skewed hash value



(a) DRAM (b) NVM-200ns/200ns (c) NVM-200ns/600ns (d) NVM-200ns/1000ns

Figure 10: Insertion latency of different hashing schemes in DRAM and NVM with different read/write latencies.

distribution without significantly affecting the maximum load factor of hash table.

In the following experiments, we set 4 slots per buckets for BCH, PFHT and level hashing, like existing work [13, 22, 25].

4.2.2 Insertion Latency

To evaluate the insertion latency of different hashing schemes, we insert unique key-value items to empty BCH, PFHT, level and path hash tables until reaching their maximum load factors. In the meantime, we measure the average latency of each insertion operation when hash tables are in the different load factors. We evaluate these hashing schemes on both DRAM and the persistent memory with different read/write latencies, i.e., 200ns/200ns, 200ns/600ns, and 200ns/1000ns. On persistent memory, these hash tables are implemented with data consistency guarantee as described in Section 4.1.

Figure 10a shows the average latency of each insertion operation in different hash tables in DRAM. Figures 10b, 10c and 10d show the average insertion latency of different hash tables in persistent memory. Compared with the experimental results in Figures 10a and 10b, we observe that the insertion latency in persistent memory is much higher than that in DRAM, while the read/write latency of persistent memory (200ns) is close to that of DRAM (136ns). The main reason is that each inserted item must be flushed into persistent memory via `CLFLUSH`, and the ordering of writes is ensured via `MFENCE` for consistency guarantee, significantly increasing the latency.

As shown in Figure 10, with the increase of the load factors, the insertion latency of BCH sharply increases, due to causing many eviction operations to deal with hash collisions. The insertion performance of BCH becomes worse in persistent memory, since the eviction operations in BCH cause many cache line flushes and memory fences. The insertion latency of PFHT increases since many items need to be inserted in the stash when the load factor is high. PFHT uses the chained hash table to manage the items in the stash. An insertion in the stash needs to allocate the node space and revise pointers, causing extra writes. The insertion latency of path hashing is higher than that of PFHT in DRAM as shown in Figure 10a, while becoming lower than that of PFHT in persistent memory as shown in Figure 10b, for a high load factor (e.g., $\geq$ 0.7). The reason is that path hashing performs only multiple read operations to find an empty bucket for inserting an item without extra write operations. Reads are much cheaper than writes in persistent memory. In both DRAM and persistent memory, level hashing has the best insertion performance due to probing fewer buckets than path hashing and rarely causes extra writes. From Figure 10b, we observe when the load factor is larger than 0.8, level hashing reduces the insertion latency by over 67%, 43%, and 30%, i.e., speeding up the insertions by over 3.0×, 1.8×, and 1.4×, compared with BCH, PFHT and path hashing.

4.2.3 Update Latency

We investigate the update latency of different hash tables with different load factors in persistent memory. The read/write latency of NVM is 200ns/600ns. As shown in Figure 11, we observe that the update latencies of BCH, PFHT, and path hashing are similar since the update only operates on a single key-value item. In a low load factor (e.g., $<$ 0.5), their update latency are significantly higher than their insertion latency as shown in Figure 10c, since each update operation needs to use the expensive logging to guarantee consistency.

To show the efficiency of our proposed opportunistic log-free update scheme as presented in Section 3.3(4), we also evaluate the update latency of *Level w/o Opp* which indicates the level hashing without this opportunistic scheme. Compared with BCH, PFHT, path hashing, and Level w/o Opp, we observe that level hashing efficiently reduces the update latency by 15% $\sim$ 52%, i.e., speeding up the updates by 1.2× $\sim$ 2.1×.

4.2.4 Search Latency

We evaluate the performance of both positive and negative searches in different hash tables on the persistent memory. For a search operation, if the target item is found in the hash table, the query is positive. Otherwise,

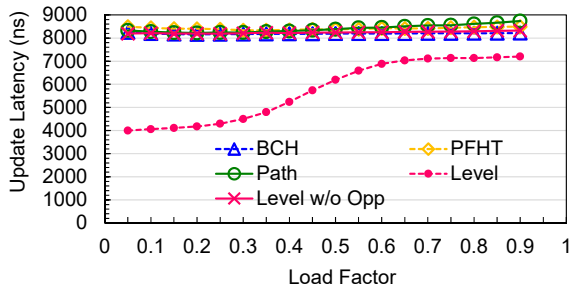


Figure 11: Average update latency of different hashing schemes in NVM.

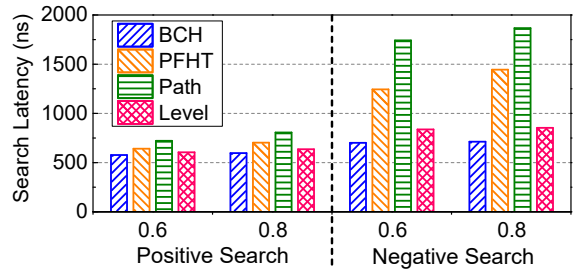


Figure 12: Average latency of positive and negative searches in level hashing.

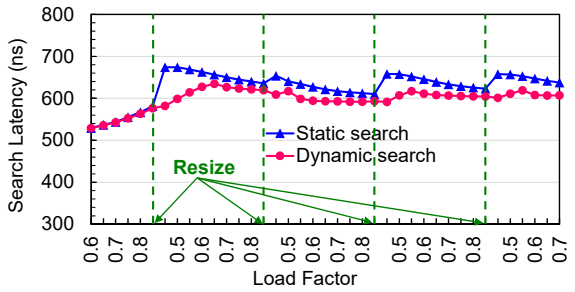


Figure 13: Average search latency of level hashing before and after resizing.

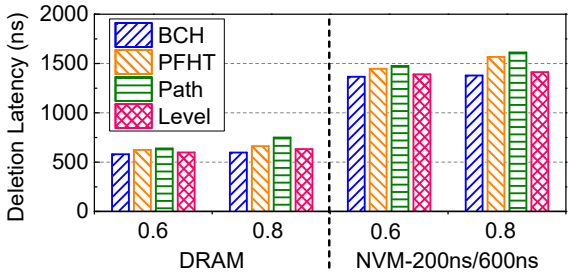


Figure 14: Average deletion latency of different hashing schemes in DRAM and NVM.

it is negative. When hash tables are in two typical load factors, i.e., 0.6 and 0.8 [68], we perform 1 million positive and negative searches respectively and measure their average latency, as shown in Figure 12.

We observe that higher load factor results in higher search latency for each hash table. Among these hash tables, BCH has the lowest positive search latency due to probing the fewest positions to find a target item. The positive search latency of level hashing is very close to that of BCH since level hashing probes at most two buckets in the bottom level when failing to find the target item in the top level. PFHT has higher positive search latency than BCH and level hashing, due to linearly searching the stash when failing to find the target item in the main hash table. The chains in the stash become long when the load factor is high, e.g., 0.8. Path hashing has the highest search latency due to probing multi-level buckets. Moreover, the negative search has higher search latency than the positive search for each hash table, since the negative search must traverse all positions that the target item may be stored. Level hashing probes at most four buckets for each search operation, which has the constant worst-case search time complexity like BCH. Nevertheless, PFHT uses chained hashing to manage the items in the stash with the $O(N_1)$ worst-case search time complexity [32], where N_1 is the number of items in the stash. The path hash table has about $\log(N_2)/2$ levels, thus producing the $O(\log(N_2))$ worst-case search time complexity, where N_2 is the total number of buckets.

To show the effectiveness of the proposed dynamic search scheme in Section 3.2(3), we evaluate the average latency of positive searches before and after resizing in level hashing. We insert unique keys into the level hash table and resize the hash table when its load factor reaches 0.85, until the level hash table is resized four times. When the level hash table is in different load factors, we perform 1-million uniform random searches. The average search latency is shown in Figure 13. We observe the search latency using the static search sharply increases after each resizing since most items are in the bottom level at this point. By performing the dynamic search, we efficiently reduce the search latency of the hash table after the first resizing.

4.2.5 Deletion Latency

We investigate the deletion latency of different hash tables in DRAM and persistent memory, as shown in Figure 14. In DRAM, we observe that the deletion latency of each hash table is approximate to its search latency since the deletion operation first searches the position storing the target item and then sets the position to null. The set-null operation has very low latency in DRAM due to being completed in CPU caches. But in persistent memory, the set-null operation causes high latency since the modified data have to be flushed into NVM for consistency guarantee. Like the positive search performance, BCH and level hashing have better deletion performance than PFHT and path hashing.

4.2.6 Resizing Time

To evaluate the resizing performance of different hashing schemes, we resize the hash tables when their load factors reach the same threshold, i.e., 0.85 (the maximum load factor that the 4-slot BCH can achieve as shown in Figure 8). We measure the total time that different hashing schemes complete the resizing. In order to show the benefit of our proposed in-place resizing scheme, we also evaluate the resizing performance of Level-Trad, which indicates the level hashing using the traditional resizing scheme [48], as shown in Figure 15.

We observe that the level hashing reduces the resizing total time by about 76%, i.e., speeding up the resizing by 4.3×, compared with Level-Trad. The reason is that the level hashing by using the in-place resizing scheme only needs to rehash the key-value items in the bottom level, significantly reducing the number of rehashed items. The number of buckets in the bottom level is 1/3 of all buckets. An item is stored in the bottom level only when both buckets in the top level are full. Hence, the items in the bottom level to be rehashed are always less than 1/3 of all items in the level hash table. Moreover, BCH, PFHT, path hashing and Level-Trad have the similar resizing time, since they need to rehash all items from the old hash table to the new one.

4.2.7 Concurrent Throughput

Since PFHT and path hashing do not support the concurrent access, we compare the concurrent level hashing with the state-of-the-art concurrent hash table in DRAM, i.e., libcuckoo [6, 37]. We focus on general hashing schemes without special hardware support. We hence use the libcuckoo with fine-grained locking instead of that with hardware transaction memory (HTM). We vary the number of concurrent threads from 2 to 16 and use the YCSB workloads with different search/insertion ratios. We use the default configuration of YCSB, i.e., zipfian request distribution with 0.99 skewness. The experimental results are shown in Figure 16. We observe that the concurrent level hashing has 1.6× −2.1× higher throughput than libcuckoo in all workloads. This is because libcuckoo incurs iterative eviction operations during an insertion. Thus an insertion needs to lock an entire cuckoo path [37], i.e., locking all slots in the eviction sequence. As a result, all insertion and search operations in other threads that access any one slot in the locked cuckoo path have to wait until the current insertion completes, thus reducing the concurrent performance. Unlike libcuckoo, in the concurrent level hashing, most insertions lock only one slot and a few insertions lock at most two slots, reducing the concurrent conflictions and thus delivering high performance.

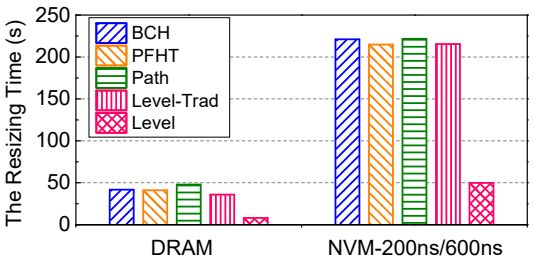


Figure 15: The resizing time of different hashing schemes in DRAM and NVM.

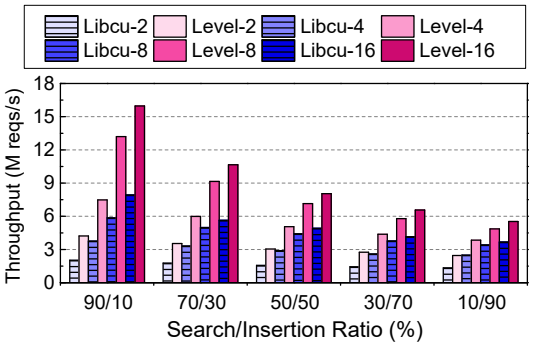


Figure 16: The concurrent request throughputs of level hashing and libcuckoo with 2/4/8/16 threads in DRAM.

5 Related Work

Tree-based Index Structures on NVM. For tree-based index structures, most work focuses on B-tree [30]. Chen et al. [16] propose a PCM-friendly B$^+$-tree that reduces PCM writes by allowing leaf nodes to be unsorted, without considering the data consistency of B$^+$-tree in PCM. Venkataraman et al. [58] propose the CDDS B-tree that leverages versioning and `CLFLUSH` and `MFENCE` instructions to achieve data consistency in B-tree. Yang et al. [64] propose the NV-Tree to guarantee the consistency of only leaf nodes in B$^+$-tree while relaxing that of internal nodes. The internal nodes can be rebuilt based on leaf nodes in case of system failures. NV-Tree reduces the number of cache line flushes due to only persisting the leaf nodes. Chen et al. [17] propose a write-atomic B-tree (wB$^+$-Tree) that adds a bitmap in each node of B$^+$-tree and achieves consistency via the atomic update of the bitmap. However, wB$^+$-Tree requires expensive redo logging for node split operations. Oukid et al. [46] propose the FP-tree, a persistent B-Tree for hybrid DRAM-NVM main memory, in which only the leaf nodes of B$^+$-tree are persisted in NVM while the internal nodes are stored in DRAM. Hwang et al. [30] propose the log-free failure-atomic shift (FAST) and in-place rebalance (FAIR) algorithms for B$^+$-tree in persistent memory via tolerating transient inconsistency. Except B-tree, Lee et al. [35] focus on the radix tree on persistent memory and propose Write Optimal Radix

Trees (WORT) that achieve data consistency via 8-byte atomic writes. Unlike them, our paper focuses on the hashing-based index structure on NVM.

Hashing-based Index Structures on NVM. Existing work on hashing-based index structures for NVM, such as PFHT [22] and path hashing [68, 69], mainly focuses on reducing NVM writes without considering the consistency issue on NVM. Unlike them, our proposed level hashing guarantees the consistency of hash table via (opportunistic) log-free schemes without expensive logging/CoW mechanisms in most cases, while delivering high performance and rarely incurring extra NVM writes. Moreover, we observe that the resizing in hash tables is expensive for the endurance and performance of NVM systems, which however is overlooked by existing work. Our paper proposes a cost-efficient in-place resizing scheme to significantly reduce the NVM writes and alleviate performance penalty during resizing.

Concurrent Hashing Index Structures. MemC3 [25] proposes an optimistic concurrent cuckoo hashing that is optimized for the multi-reader and single-writer concurrency by using a global lock and version counters. The Intel Threading Building Blocks (TBB) [3] provides a chaining-based concurrent hash table using per-bucket fine-grained locking. Libcuckoo [37] is a multi-reader and multi-writer concurrent cuckoo hashing scheme using fine-grained locking that delivers higher performance than the TBB hash table. Our proposed concurrent level hashing has higher concurrent throughput than libcuckoo due to locking fewer slots for insertions. To support variable-length keys and values, MemC3 [25] stores a short summary of the key and a pointer for each key-value item in the hash table. This pointer points to the full key-value term that is stored outside the hash table. The same method can be added into level hashing as needed to support variable-length keys and values.

6 Conclusion

In order to efficiently index the data on persistent memory, this paper proposes a write-optimized and high-performance hashing index scheme, called level hashing, along with a cost-efficient in-place resizing scheme and (opportunistic) log-free consistency guarantee schemes. Level hashing efficiently supports multi-reader and multi-writer concurrency via simply using fine-grained locking. We have evaluated level hashing in both DRAM and NVM platforms. Compared with the state-of-the-art hashing schemes, level hashing achieves $1.4\times-3.0\times$ speedup for insertions, $1.2\times-2.1\times$ speedup for updates, and over $4.3\times$ speedup for resizing while maintaining high search and deletion performance. Compared with the start-of-the-art concurrent hashing scheme, the concurrent level hashing improves the throughput by $1.6\times-2.1\times$.

Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, and National Natural Science Foundation of China (NSFC) under Grant 61772212. We are grateful to our shepherd, Steven Swanson, and the anonymous reviewers for their constructive feedback and suggestions.

References

[1] Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products. https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/, 2015.

[2] Quartz: A DRAM-based performance emulator for NVM. https://github.com/HewlettPackard/quartz, 2015.

[3] Intel® Threading Building Blocks. https://www.threadingbuildingblocks.org/, 2017.

[4] Intel® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/en-us/isa-extensions, 2018.

[5] JAVA HashMap. http://www.docjar.com/html/api/java/util/HashMap.java.html, 2018.

[6] Libcuckoo: A high-performance, concurrent hash table. https://github.com/efficient/libcuckoo, 2018.

[7] Memcached. https://memcached.org/, 2018.

[8] Redis. https://redis.io/, 2018.

[9] AKINAGA, H., AND SHIMA, H. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE 98*, 12 (2010), 2237–2251.

[10] APALKOV, D., KHVALKOVSKIY, A., WATTS, S., NIKITIN, V., TANG, X., LOTTIS, D., MOON, K., LUO, X., CHEN, E., ONG, A., DRISKILL-SMITH, A., AND KROUNBI, M. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC) 9*, 2 (2013), 13.

[11] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[12] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[13] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., AND TULLSEN, D. M. Horton tables: Fast hash tables for in-memory data-intensive computing. In *USENIX Annual Technical Conference (USENIX ATC)* (2016).

[14] BYERS, J., CONSIDINE, J., AND MITZENMACHER, M. Simple load balancing for distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)* (2003).

[15] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[16] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)* (2011).

[17] CHEN, S., AND JIN, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.

[18] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[19] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[20] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009).

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)* (2010).

[22] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)* (2015).

[23] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (1984).

[24] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014).

[25] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[26] GAO, H., GROOTE, J. F., AND HESSELINK, W. H. Almost wait-free resizable hashtables. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)* (2004).

[27] GARCIA-MOLINA, H., AND SALEM, K. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering 4*, 6 (1992), 509–516.

[28] GUERRA, J., MÁRMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *USENIX Annual Technical Conference (USENIX ATC)* (2012).

[29] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing (DISC)* (2008).

[30] HWANG, D., KIM, W.-H., WON, Y., AND NAM, B. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[31] KIM, W.-H., KIM, J., BAEK, W., NAM, B., AND WON, Y. NVWAL: exploiting nvram in write-ahead logging. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[32] KNUTH, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[33] KOCBERBER, O., GROT, B., PICOREL, J., FALSAFI, B., LIM, K., AND RANGANATHAN, P. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013).

[34] LAI, C., JIANG, S., YANG, L., LIN, S., SUN, G., HOU, Z., CUI, C., AND CONG, J. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)* (2015).

[35] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2017).

[36] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015).

[37] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014).

[38] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)* (2017).

[39] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., ZHENG, W., AND REN, J. DUDETM: building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[40] LIU, Y., ZHANG, K., AND SPEAR, M. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing (PODC)* (2014).

[41] LIU, Z., CALCIU, I., HERLIHY, M., AND MUTLU, O. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2017).

[42] MAURER, W. D., AND LEWIS, T. G. Hash table methods. *ACM Computing Surveys (CSUR) 7*, 1 (1975), 5–19.

[43] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[44] MUELLER, W., AICHMAYR, G., BERGNER, W., ERBEN, E., HECHT, T., KAPTEYN, C., KERSCH, A., KUDELKA, S., LAU, F., LUETZEN, J., ORTH, A., NUETZEL, J., SCHLOESSER, T., SCHOLZ, A., SCHROEDER, U., SIECK, A., SPITZER, A., STRASSER, M., WANG, P.-F., WEGE, S., AND WEIS, R. Challenges for the dram cell scaling to 40nm. In *IEEE International Electron Devices Meeting (IEDM)* (2005).

[45] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[46] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2016).

[47] PAGH, R., AND RODLER, F. F. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms (ESA)* (2001).

[48] PIGGIN, N. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. linux kernel mailing list. https://lwn.net/Articles/302132/, 2008.

[49] PITTEL, B. Linear probing: the probable largest search time grows logarithmically with the number of records. *Journal of algorithms 8*, 2 (1987), 236–249.

[50] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2009).

[51] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. K. Log-structured memory for DRAM-based storage. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2014).

[52] SEO, J., KIM, W.-H., BAEK, W., NAM, B., AND NOH, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[53] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM 53*, 3 (2006), 379–405.

[54] SHUN, J., AND BLELLOCH, G. E. Phase-concurrent hash tables for determinism. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2014).

[55] SUN, Y., HUA, Y., JIANG, S., LI, Q., CAO, S., AND ZUO, P. Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)* (2017).

[56] THOZIYOOR, S., AHN, J. H., MONCHIERO, M., BROCKMAN, J. B., AND JOUPPI, N. P. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture (ISCA)* (2008).

[57] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference (USENIX ATC)* (2011).

[58] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2011).

[59] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2015).

[60] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[61] WONG, H.-S. P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

[62] XIA, F., JIANG, D., XIONG, J., AND SUN, N. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2017).

[63] XU, J., AND SWANSON, S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2016).

[64] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: reducing consistency cost for nvm-based single level systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2015).

[65] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment 8*, 3 (2014), 209–220.

[66] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[67] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2009).

[68] ZUO, P., AND HUA, Y. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)* (2017).

[69] ZUO, P., AND HUA, Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (2018), 985–998.

FLASHSHARE: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs

Jie Zhang[1], Miryeong Kwon[1], Donghyun Gouk[1], Sungjoon Koh[1], Changlim Lee[1],
Mohammad Alian[2] , Myoungjun Chun[3], Mahmut Taylan Kandemir[4],
Nam Sung Kim[2], Jihong Kim[3], and Myoungsoo Jung[1]

Yonsei University[1],
Computer Architecture and Memory Systems Laboratory,
University of Illinois Urbana-Champaign[2], Seoul National University[3], Pennsylvania State University[4]
http://camelab.org

Abstract

A modern datacenter server aims to achieve high energy efficiency by co-running multiple applications. Some of such applications (e.g., web search) are latency sensitive. Therefore, they require low-latency I/O services to fast respond to requests from clients. However, we observe that simply replacing the storage devices of servers with Ultra-Low-Latency (ULL) SSDs does not notably reduce the latency of I/O services, especially when co-running multiple applications. In this paper, we propose FLASHSHARE to assist ULL SSDs to satisfy different levels of I/O service latency requirements for different co-running applications. Specifically, FLASHSHARE is a holistic cross-stack approach, which can significantly reduce I/O interferences among co-running applications at a server without any change in applications. At the kernel-level, we extend the data structures of the storage stack to pass attributes of (co-running) applications through all the layers of the underlying storage stack spanning from the OS kernel to the SSD firmware. For given attributes, the block layer and NVMe driver of FLASHSHARE differently manage the I/O scheduler and interrupt handler of NVMe. We also enhance the NVMe controller and cache layer at the SSD firmware-level, by dynamically partitioning DRAM in the ULL SSD and adjusting its caching strategies to meet diverse user requirements. The evaluation results demonstrate that FLASHSHARE can shorten the average and 99^{th}-percentile turnaround response times of co-running applications by 22% and 31%, respectively.

1 Introduction

Datacenter servers often run a wide range of online applications such as web search, mail, and image services [8]. As such applications are often required to satisfy a given Service Level Agreement (SLA), the servers should process requests received from clients and send the responses back to the clients within a certain amount of time. This requirement makes the online applications latency-sensitive, and the servers are typically (over)provisioned to meet the SLA even when they unexpectedly receive many requests in a short time period. However, since such events are infrequent, the average utilization of the servers is low, resulting in low energy efficiency with poor energy proportionality of contemporary servers [28, 17].

To improve utilization and thus energy efficiency, a server may run an online application with offline applications (e.g., data analytics workloads), which are latency-insensitive and are often throughput-oriented [26, 30, 29]. In such cases, it becomes challenging for the server to satisfy a given SLA for the online application because co-running these applications further increase I/O service latency. We observe that device-level I/O service latency of a high-performance NVMe solid state drive (SSD) contributes to more than 19% of the total response time of online applications, on average. To reduce the negative impact of long I/O service latency on response time of online applications, we may deploy Ultra-Low-Latency (ULL) SSDs based on emerging memory, such as Z-NAND [36] or 3D-Xpoint [15]. These new types of SSDs can accelerate I/O services with ULL capability. Our evaluation shows that ULL SSDs (based on Z-NAND) can give up to 10× shorter I/O latency than the NVMe SSD [14] (cf. Section 2).

These ULL SSDs offer memory-like performance, but our in-depth analysis reveals that online applications cannot take full advantage of ULL SSDs particularly when a server co-runs two or more applications for higher utilization of servers. For example, the 99^{th} percentile re-

sponse time of *Apache* (i.e., online application) is 0.8 *ms*. However, the response time increases by 228.5% if the server executes it along with a *PageRank* (i.e., offline application). A reason behind this offset of the benefits of memory-like performance is that server's storage stack lacks understanding of criticality of user's I/O requests and its impact on the response time or throughput of a given application.

In this paper, we propose FLASHSHARE, a holistic cross-stack approach that enables a ULL SSD device to directly deliver its low-latency benefits to users and satisfy different service-level requirements. Specifically, FLASHSHARE fully optimizes I/O services from their submission to execution to completion, by punching through the current server storage stack. To enable this, FLASHSHARE extends OS kernel data structures, thereby allowing users to dynamically configure their workload attributes (for each application) without any modification to their existing codes. FLASHSHARE passes these attributes through all components spanning from kernel to firmware and significantly reduces inter-application I/O interferences at servers when co-running multiple applications. The specific stack optimizations that this work performs can be summarized as follows:

• *Kernel-level enhancement.* At the kernel-level, there are two technical challenges in exposing the pure performance of ULL SSDs to users. First, the Linux multi-queue block layer (blk-mq) holds I/O requests in its software/hardware queues, introducing long latencies. Second, the current standard protocol of the NVMe queuing mechanism has no policy on I/O prioritization, and therefore, a request from an offline application can easily block an urgent I/O service requested by an online application. FLASHSHARE carefully bypasses the latency-critical requests to the underlying NVMe queue. In addition, our NVMe driver pairs NVMe submission and completion queues by being aware of the latency criticality (per application).

• *Firmware-level design.* Even though kernel-level optimizations guarantee to issue latency-critical requests with the highest order, the ULL characteristics (memory-like performance) cannot be fully exposed to users if the underlying firmware has no knowledge of latency criticality. In this work, we redesign the firmware for I/O scheduling and caching to directly disclose ULL characteristics to users. We partition ULL SSD's embedded cache and separately allocate the cache for each I/O service based on its workload attributes. Our firmware dynamically updates the partition sizes and adjusts the prefetch I/O granularity in a fine-granular manner.

• *New interrupt services for ULL SSDs.* We observe that the current NVMe interrupt mechanism is not optimized for ULL I/O services, due to the long latency incurred by storage stack layers. We also discover that a polling method (implemented in Linux 4.9.30) consumes many CPU cycles to check the completion of I/O services, which may not be a feasible option for servers co-running two or more applications. FLASHSHARE employs a selective interrupt service routine (Select-ISR), which uses message-signaled interrupts for only offline applications, while polling the I/O completion for online interactive applications. We further optimize the NVMe completion routine by offloading the NVMe queue and ISR management into a hardware accelerator.

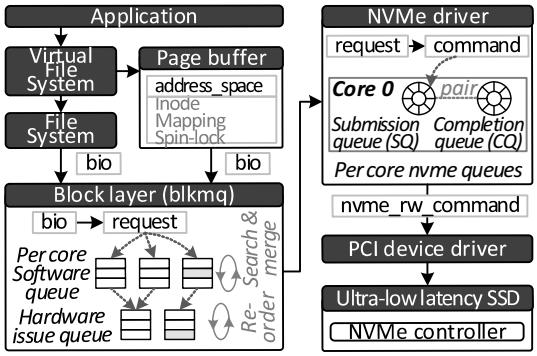


Figure 1: High-level view of software kernel stack.

We implement the kernel enhancement components in a real I/O stack of Linux, while incorporating Select-ISR and hardware/firmware modifications using a full system simulation framework [2, 21]. We also revise the memory controller and I/O bridge model of the framework, and validate the simulator with a real 800GB Z-SSD prototype. The evaluation results show that FLASHSHARE can reduce the latency of I/O stack and the number of system context switch by 73% and 42%, respectively, while improving SSD internal cache hit rate by 37% in the co-located workload execution. These in turn shorten the average and 99^{th} percentile request turnaround response times of the servers co-running multiple applications (from an end-user viewpoint) by 22% and 31%, respectively.

2 Background

2.1 Storage Kernel Stack

Figure 1 illustrates the generic I/O stack in Linux, from user applications to low-level flash media. An I/O request is delivered to a file system driver through the virtual file system interface. To improve system-level performance, the request can be buffered in the page buffer module, using an `address_space` structure, which in-

cludes the `inode` information and mapping/spin-lock resources of the owner file object. When a cache miss occurs, the file system retrieves the actual block address, referred to as Logical Block Address (LBA) by looking up `inodes` and sends the request to the underlying multi-queue block layer (blk-mq) through a `bio` structure.

In contrast to the kernel's block layer that operates with a single queue and lock, the multi-queue block layer (blk-mq) splits the queue into multiple separate queues, which helps to eliminate most contentions on the single queue and corresponding spin-lock. blk-mq allocates a `request` structure (associated to `bio`) with a simple tag and puts it in the per-CPU software queues, which are mapped to the hardware issue queues. The software queue of blk-mq merges the incoming `request` with an already-inserted `request` structure that has the same LBA, or an adjacent LBA to the current LBA. The merge operation of blk-mq can reduce the total number of I/O operations, but unfortunately, it consumes many CPU cycles to search through the software queues. From the latency viewpoint, the I/O merging can be one of the performance bottlenecks in the entire storage stack. On the other hand, the hardware issue queues simply buffer/reorder multiple requests for the underlying NVMe driver. Note that the hardware issue queue can freely reorder the I/O requests without considering the I/O semantics, since the upper-level file system handles the consistency and coherence for all storage requests.

The NVMe driver exists underneath blk-mq, and it also supports a large number of queue entries and commands per NVMe queue. Typically, each deep NVMe queue is composed of pairing a *submission queue (SQ)* and a *completion queue (CQ)*. The NVMe driver informs the underlying SSD of the arrivals and completions of I/O requests through head and tail pointers, allocated per NVMe queue. In the storage stack, every request issued by the NVMe driver is delivered to the PCI/PCIe device driver in the form of a `nvme_rw_command` structure, while the SSD dispatches them in an active manner; in contrast to other storage protocols in which a host-side controller must dispatch or transfer all data and commands, NVMe SSDs can pull the command and data stored in system memory from storage side without a host intervention. When the I/O request is completed by the SSD, it sends a *message signaled interrupt (MSI)* that directly writes the interrupt vector of each core's programmable interrupt controller. The interrupted core executes an ISR associated with the vector's interrupt request (IRQ). Subsequently, the NVMe driver cleans up the corresponding entry of the target SQ/CQ and returns the completion results to its upper layers, such as blk-mq and filesystem.

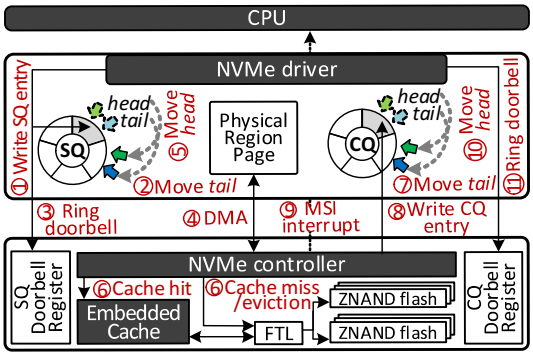


Figure 2: Overview of device firmware stack.

2.2 Device Firmware Stack

Based on the NVMe specification, the deep queues are created and initialized by the host's NVMe driver through the administrator queues, and the I/O requests in the queues are scheduled by the NVMe controller that exists on top of the NVMe SSD firmware stack [46]. Most high-performance SSDs, including all devices we tested in this study [13, 14, 36], employ a large internal DRAM (e.g., 1 GB $\sim$ 16 GB). Thus, underneath the NVMe controller, SSDs employ an embedded cache layer, which can immediately serve the I/O requests from the internal DRAM without issuing an actual storage-level operation when a cache hit occurs at the internal DRAM [42, 20]. If a cache miss or replacement is observed, the NVMe controller or cache layer generates a set of requests (associated with miss or replacement) and submits them to the underlying flash translation layer (FTL), which manages many Z-NAND chips across multiple channels [6, 18].

Figure 2 shows the components of the firmware stack and depicts how the NVMe controller pulls/pushes a request to/from the host. Specifically, when the NVMe driver receives a `request` (①), it increases the tail/head of SQ (②) and writes the doorbell register (③) that the NVMe controller manages. The NVMe controller then initiates to transfer the target data (④) associated with the tail from the host's kernel memory pointed by the corresponding Physical Region Page (PRP) (stored in `nvme_rw_command`). Once the DMA transfer is completed, the NVMe controller moves the head to the NVMe queue entry pointed by the tail (⑤), and forwards the request to either the embedded cache layer or underlying FTL (⑥). When a cache miss or replacement occurs, the FTL translates the target LBA to the corresponding physical page address of the underlying Z-NAND, and performs complex flash-related tasks (if needed), such as garbage collection and wear-leveling

Components	Spec.	Components	Spec.
CPU	i7-4790	Memory	32GB
	3.6GHz		DDR3
	8 cores	Chipset	H97

Table 1: Server configurations.

[38, 33, 5, 4, 12]. Unlike traditional NAND [11], Z-NAND completes a 4KB-sized read service within 3 μs [36] and we observed that a Z-NAND based ULL SSD can complete an I/O service within 47$\sim$52 μs, including data transfer and FTL execution latencies (cf. Figure 3a).

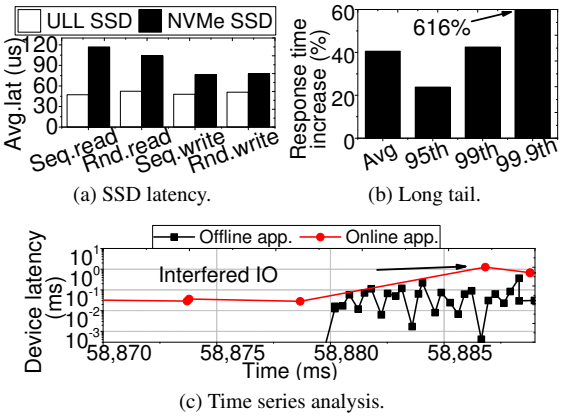
After completing the service of the I/O request, the NVMe controller increases the corresponding tail pointer of CQ (⑦). It then performs a DMA transfer to the host and changes the phase tag bit associated with the target CQ entry (⑧). The controller notifies the DMA completion by sending an MSI to the host (⑨). The host's ISR checks the phase tag by searching through the queue entries from the head to the tail. For the ones that have a valid phase tag, the ISR clears the tag bit and processes the rest of the I/O completion routines. Finally, it increases the head of CQ (⑩), removes the corresponding entry of SQ, and writes the CQ's head doorbell of the NVMe controller (⑪). While polling is not a standard method in the NVMe specification, the state-of-the-art Linux (4.9.30) can support it, since the NVMe controller directly changes the phase tags of the target entries over PCIe before sending the corresponding MSI. Thus, in the kernel storage stack, the NVMe driver checks the phase tags of CQ and simply ignores the MSI updates.

3 Cross-Layer Design

3.1 Challenges with Fast Storage

In this section, we characterize the device latency of a prototype of real 800GB Z-NAND based ULL SSD by comparing it against the latency of a high-performance NVMe SSD [14]. We then evaluate the performance of a server equipped with the ULL SSD, when *Apache* (an online latency-sensitive application) co-runs with *PageRank* (an offline throughput-oriented application). While *Apache* requires responding to the service coming from the client through TCP/IP by retrieving data on object storage, *PageRank* performs data analytics over Hadoop's MapReduce (24GB dataset). The configuration details of the server under test are listed in Table 1.

Figure 3a compares the average latency of the ULL SSD and NVMe SSD, with the number of queue entries varying from 1 to 64. The latencies of the ULL SSD for the random and sequential access patterns are 42% and 48% shorter than that of the NVMe SSD, respectively. However, as shown in Figure 3b, we observe that the turnaround response times of the online application significantly degrade when co-running it along with the offline application. Specifically, the response time of *Apache* becomes 41% longer if *PageRank* also runs on the same server. This performance deterioration is also observed in the long tail: the 95^{th} and 99^{th} response times of *Apache* under the co-running scenario increase by 24% and 43%, respectively, compared to those of an *Apache*-only execution scenario.



(a) SSD latency. (b) Long tail.

(c) Time series analysis.

Figure 3: Application co-running analysis.

The reason behind these response time increases is captured by Figure 3c. Once *PageRank* begins to perform I/O services (at 58,880 ms), the I/O services of *Apache* gets interfered by *PageRank*, and this increases the response time of *Apache* by 42$\times$ compared to the standalone execution situation (before 58,880 ms). This happens because the server storage stack has no knowledge of the ultra-low latency exposed by the underlying Z-NAND media, and also most of the components in the stack cannot differentiate *Apache*'s I/O services from *PageRank*'s I/O services (even though the two applications require different levels of the I/O responsiveness).

3.2 Responsiveness Awareness

It is very challenging for the kernel to speculate workload behaviors and predict the priority/urgency of I/O requests [24]. Since users have a better knowledge of I/O responsiveness, a more practical option is to offer a set of APIs to users. However, such APIs require significant changes to existing server application's sources. Instead, we modify the Linux process control block, called `task_struct`, to accommodate a *workload attribute* for each application. A potential issue in leveraging the attribute, stored in `task_struct`, from the software layers in the storage stack is that a reference of `task_struct` may not be valid, based on the loca-

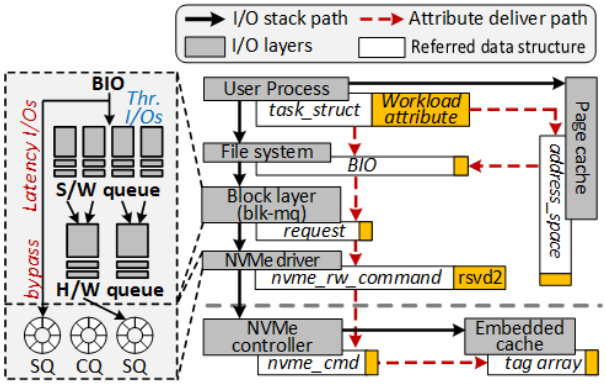


Figure 4: Overview of our server stack optimizations.

tion of the storage stack and the timing when a layer retrieves such `task_struct`. Therefore, it is necessary for blk-mq and NVMe driver to have their own copies of the workload attribute per I/O service. To this end, we further extend `address_space`, `bio`, `request`, and `nvme_rw_command` structures to punch through the storage stack and pass the workload attribute to the underlying SSD firmware.

As such, FLASHSHARE provides a utility, called chworkload_attr, which allows servers to configure and dynamically change the attribute of each application similar as the nice mechanism [9]. The chworkload_attr helps users to embed the criticality of responsiveness into each application's `task_struct`. We modify the system call table (e.g., `arch/x86/entry/syscalls/syscall_64.tbl`) and implement two system calls, to set/get the workload attribute to/from the target `task_struct`. These system invocations are registered at `/linux/syscall.h` with the `asmlinkage` tag. They change the attribute of a specific process (given by the user from a shell), which is implemented in `/sched/cores.c`. The chworkload_attr simply invokes the two system calls with an appropriate system call index, registered in the system table. Using such interfaces, the chworkload_attr can capture the attribute and fill the information in `task_struct` for each application at the kernel level. It should be noted that the chworkload_attr is designed for server-side users (e.g., datacenter operators), not for client-side users who may recklessly ask a higher priority all the time.

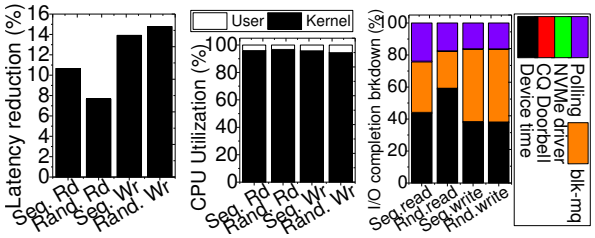
Figure 4 illustrates how the workload attribute is referred by `task_struct` in the storage stack modified by FLASHSHARE. If an incoming I/O request uses a direct I/O (`O_DIRECT`), the file system driver (EXT4 used in our implementation) retrieves the attribute and puts it into `bio`. Otherwise, the page cache copies the attribute from `task_struct` to `address_space`. Therefore, when `blk-mq` receives a `bio`, it includes the criticality of responsiveness in the attribute, and copies that information to a `request` structure. Lastly, the NVMe driver overrides the attribute to an unused field, called `rsvd2` of `nvme_rw_command`. The underlying SSD's firmware can catch the host-side attribute information per request by reading out the value in `rsvd2` and passing it to the NVMe controller and embedded cache layer, the details of which will be explained in Section 4.

3.3 Kernel Layer Enhancement

By utilizing the workload attributes, we mainly optimize the two layers underneath the file system: blk-mq and NVMe driver, as shown in Figure 4. The software and hardware queues of blk-mq hold I/O requests with the goal of merging or reordering them. Even though a deep blk-mq queue can increase chances for merging and reordering requests thereby higher bandwidth utilization, it also introduces long queue waiting delays. This can, unfortunately, hurt the responsiveness of online applications (and cannot take the advantage of ULL). To address this potential shortcoming, we enhance blk-mq to bypass all the I/O services requested from the online application to the NVMe driver (without queueing), while tracking other requests coming from the throughput applications just like normal software and hardware queues. However, this simple bypass strategy potentially raises an I/O hazard issue; a hazard could happen if an offline application has an I/O request being scheduled by blk-mq to the same LBA that a subsequent online application issued.

Because such request cannot be skipped in the queue, blk-mq retrieves it, which may have the potential hazard, from the software queue. If the operation type of the retrieved request is different from that of the incoming request that we want to bypass, blk-mq submits the retrieved request along with the incoming request in tandem. Otherwise, blk-mq merges those two requests into a single `request` structure and forwards the merged request to the underlying NVMe driver.

Under blk-mq, the NVMe driver submits the bypassed request to the corresponding SQ. One of the issues in the NVMe queue management is that the head and tail pointers for a pair of the target CQ/SQ are managed by the (kernel-side) NVMe driver and the (firmware-side) NVMe controller together in a round-robin fashion. Thus, even though our modification in blk-mq prioritizes latency-critical I/O services by expecting them to be scheduled in the SQ earlier than other requests, the NVMe controller can dispatch a service requested by an offline application prior to the latency-critical I/O



(a) Average I/O latency. (b) CPU utilization. (c) I/O completion breakdown.

Figure 5: I/O execution analysis of ULL SSD.

service. This, in turn, makes the service latency of the latter considerably longer. To address this undesirable situation, we create two SQs and one CQ per core as a pair of NVMe queue, which is different from a traditional NVMe queue management strategy. Specifically, as shown in Figure 4, an SQ between two SQs is used for the requests whose attributes come from online applications. In our implementation, the NVMe driver sends a message via the administrator queue to inform the NVMe controller of selecting a new queue arbitration method that always gives a high priority to scheduling requests in such the SQ. To avoid a starvation owing to the priority SQ, the NVMe driver drains the I/O requests originating from the offline applications if the number of such queued requests is greater than a threshold, or if they are not served within a certain amount of time. We observed that it is best to start draining the queue with a 200 μs threshold or when there are 8 pending queue entries.

4 I/O Completion and Caching

Figure 5a shows the actual latency improvement when we use the Linux 4.9.30 polling mechanism for a ULL SSD (Z-SSD prototype). In this evaluation, we set the size of all the requests to 4KB. As shown in Figure 5a, the I/O latency with the polling mechanism is 12% shorter than the one managed by MSI for all I/O request patterns. However, we also observe that the cores in the kernel mode are always busy in handling I/O completions. Specifically, Figure 5b shows the CPU utilization of the polling mechanism for both the kernel and user modes. This figure shows that the CPU utilization for polling gets significantly high (almost 97% of CPU cycles are used for only polling the I/O completion). This high CPU utilization presents two technical issues. First, as there is no core to allocate in handling the criticality of I/O responsiveness, the original polling method is not a feasible option for a server co-running multiple applications. Second, while a 12% latency improvement of the polling method is still promising, we could shorten

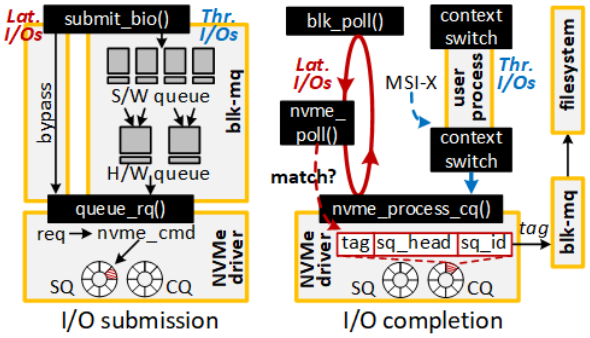


Figure 6: The process implemented by the selective interrupt service routine.

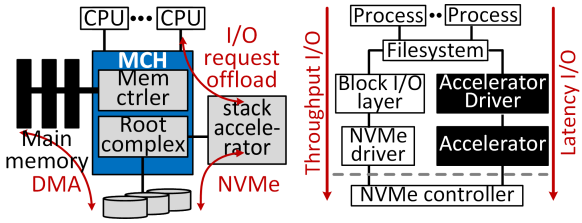
the latency even more, if we could alleviate the core-side overheads brought by polling for the I/O completion.

4.1 Selective Interrupt Service Routine

FLASHSHARE uses polling for only I/O services originating from online applications, while MSI is still used for offline applications. Figure 6 shows how this selective ISR (Select-ISR) is implemented. We change `submit_bio()` of blk-mq to insert an incoming request (i.e., `bio`), delivered by the file system or page cache, into its software queue if the attribute of `bio` indicates an offline application. This request will be reordered and served just like a normal I/O operation. In contrast, if the incoming request is associated with an online application, blk-mq directly issues it to the underlying NVMe driver by invoking `queue_rq()`. The NVMe driver then converts the I/O request into NVMe commands and enqueues it into the corresponding SQ.

With Select-ISR, the CPU core can be released from the NVMe driver through a context switch (CS), if the request came from offline applications. Otherwise, blk-mq invokes to the polling mechanism, `blk_poll()`, after recording the tag of the I/O service along with online applications. `blk_poll()` continues to invoke `nvme_poll()`, which checks whether a valid completion entry exists in the target NVMe CQ. If it is, blk-mq disables IRQ of such CQ so that MSI cannot hook the procedures of blk-mq later again. `nvme_poll()` then looks up the CQ for a new entry by checking the CQ's phase tags. Specifically, `nvme_poll()` searches an CQ entry whose request information is matched with the tag that `blk_poll()` waits for completion. Once it detects such a new entry, blk-mq exits from the infinite iteration implemented in `blk_poll()` and switches the context to its user process.

A challenge in enhancing the storage stack so that it can be aware of ULL is that, even though we propose



(a) Hardware viewpoint. (b) Software viewpoint.

Figure 7: Overview of our I/O stack accelerator.

Select-ISR, polling still wastes many CPU cycles and blk-mq consumes kernel CPU cycles to perform simple operations, such as searching the tag in SQ/CQ and merging the requests for each I/O service invocation. This is not a big issue with conventional SSDs, but with a ULL SSD, it can prevent one from enjoying the full benefits of low latency. For example, in Figure 5c, we observed that polling and storage stack modules, including ISR, context switching, and blk-mq cycles, take 58% of total I/O completion time. Thus, as a further enhancement of Select-ISR, we propose an I/O-stack accelerator. Figure 7 shows how our I/O-stack accelerator is organized from the hardware and software viewpoints. This additional enhancement migrates the management of the software and hardware queues from blk-mq to an accelerator attached to a PCIe. This allows a `bio` generated by the upper file system to be directly converted into a `nvm_rw_command`. Especially, the accelerator searches a queue entry with a specific tag index and merges `bio` requests on behalf of a CPU core. The offload of such tag search and merge operations can reduce the latency overhead incurred by the software layers in the storage stack by up to 36% of the total I/O completion time. The specifics of this accelerator are described in Section 4.3.

4.2 Firmware-Level Enhancement

In our implementation, the NVMe controller is aware of the two SQs per core, and gives a higher priority to the I/O service enqueued in the latency-critical SQ. While this I/O scheduling issue can be simply updated, a modification of the embedded cache layer to expose a shorter latency to online applications can be challenging. Specifically, the cache layer can starve latency-critical I/O services if it serves more throughput-oriented I/O services. This situation can be observed even when the cache layer understands the workload attribute brought by the NVMe driver/controller, as the internal DRAM is a shared resource in a given SSD. In addition, the I/O patterns and locality of online applications are typically different from those of offline applications. That is, a single generic

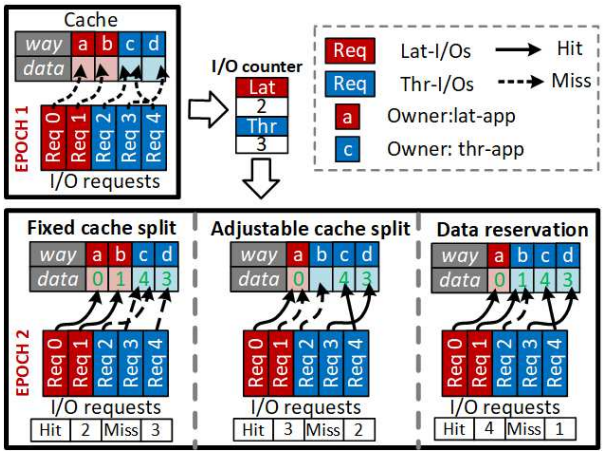


Figure 8: Example of adjustable cache partition.

cache access policy cannot efficiently manage I/O requests from both online and offline applications.

The cache layer of FLASHSHARE partitions the DRAM into two DRAM caches with the same number of sets, but different ways of associativity (i.e., cache ways) [45], and allocates each to online and offline applications, separately. If the size of partitioned caches is fixed, it can introduce cache thrashing depending on the I/O behavior of the given applications (and the corresponding workload patterns). For example, in Figure 8, if two partitioned caches (one for online applications and another for offline applications) employ two ways for each, the requests $2 \sim 4$ compete for the way 'c', and they experience cache misses.

To address this, in cases of high I/O demands, our cache layer collects the number of I/O accesses for the online and offline applications at each epoch. The proposed cache layer dynamically adjusts the number of cache ways allocated to two different cache partitions. As shown in Figure 8, if the cache splits can be adjusted (cf. adjustable cache split), the ways 'b'$\sim$'d' can accommodate the requests $2 \sim 4$. However, as the way 'b' is reallocated to the I/O requests of offline applications (e.g., throughput-oriented I/Os), the latency critical request 1 is unable to access data residing in the way 'b', introducing cache miss. To address this challenge, when adjusting the cache ways, the cache layer keeps the data for the previous owner as "read-only" until a new request is written into the corresponding way.

Most firmware in SSDs read out the data from multiple memory media to improve parallelism, and therefore, the cache can be polluted by an ill-tuned prefetching technique. As shown in Figure 9, we leverage a "ghost caching" mechanism [37, 31, 34] to help the SSD controller to evaluate the performance (i.e., cache

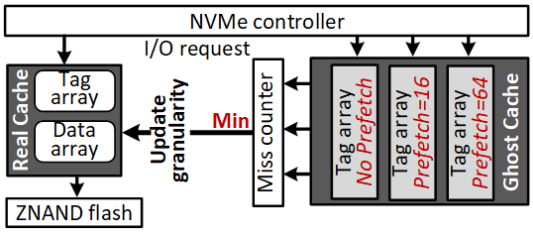


Figure 9: Adjustable cache prefetching scheme.

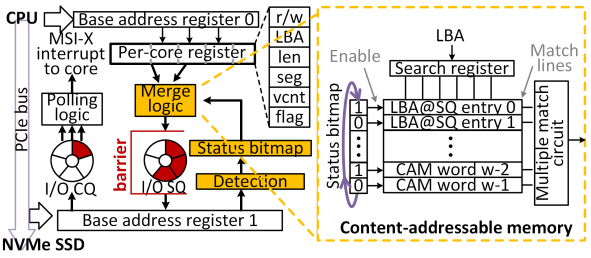


Figure 10: Design details of hardware accelerator.

miss rate) of various prefetching configurations and adjust the cache layer to the optimal prefetching configuration at runtime. Specifically, we build multiple ghost caches, each maintaining only cache tags without any actual data. The associativity and size of ghost caches are configured the same way as the configuration of the proposed cache layer, but each of these caches employs a different prefetching I/O granularity. In each epoch, FLASHSHARE identifies the ghost cache that exhibits a minimum number of cache misses, and changes the prefetch granularity of the cache layer to that of the selected ghost cache.

4.3 I/O-Stack Acceleration

We load the kernel driver of the proposed accelerator as an upper layer module of blk-mq (cf. Figure 7b). As shown in Figure 10, the accelerator driver checks if the incoming `bio` is associated with online latency-critical applications. If so, the driver extracts the operation type, LBA, I/O size, memory segment pointer (related to target data contents), and the number of memory segments from `bio`' `bio->bi_opf`, `bio->bi_iter.bi_sector`, `bio->bi_iter.bi_size`, `bio->bi_io_vec` and `bio->bi_vcnt`, respectively. The kernel driver then writes this extracted information into the corresponding registers in base address registers (BAR) of the accelerator. The accelerator then identifies an I/O submission queue entry that has an LBA, which is the same as the target LBA of incoming `bio` request. If the accelerator finds such an entry, its merge logic automatically merges the information (stored in BAR) into the target entry; otherwise, the accelerator composes a new NVMe command and appends it to the tail of the target SQ. Then, the accelerator rings (writes) the doorbell register of the underlying ULL SSD. However, as the merge logic and ULL SSD can simultaneously access the I/O SQ entries, an I/O hazard may occur. To prevent such situations, we propose to add a *barrier logic*, which is a simple MUX and works as a hardware arbitrator. It allows either the merge logic or ULL SSD (via BAR1 register) to access the target NVMe SQ at one time. Once the I/O request is inserted into the SQ, the polling logic of our accelerator starts to poll the corresponding CQ. When the I/O service is completed, the accelerator raises an interrupt signal for the accelerator driver. The driver then takes the control of I/O completion.

Note that, since searching through all entries can introduce a long latency, the accelerator employs content-addressable memory (CAM), which keeps the LBA of received `nvme_cmd` instances (for I/O submissions). Using the content-addressable memory, our accelerator in parallel compares the incoming LBA with all recorded LBAs, thereby significantly reducing the search latency. For the simultaneous comparison, the number of CAM entries is set to be same as the number of SQ entries. In addition, if there is no issued `nvme_cmd` or `nvme_cmd` instance(s) is fetched by the underlying ULL SSD, the corresponding SQ entries should be invalid. Thus, we introduce a *status bitmap* to filter the entries, which do not contain valid `nvme_cmd` instances. Specifically, our status bitmap is set to 1, if merge logic inserts a new NVMe command; the status bitmap is reset to 0, if the ULL SSD is detected to pull NVMe commands from I/O SQ. If the status bitmap indicates that the request entries in CAM (associated with the target SQ) are invalid, CAM will skip searching those entries.

5 Evaluation

5.1 Experimental Setup

Implementation environment. We simulate our holistic optimization approaches on an event-driven computer-system architecture simulator, gem5 [2]. Specifically, we configure it to a full system mode which runs on 64-bit ARM ISA. We use Linux 4.9.30 as the default kernel in this simulation, and set up 8 ARM cores with 2GHz, which have private 64KB L1 data and 64KB L1 instruction caches. We also configure a 2GB DRAM-based main memory, which is shared by all 8 cores. Note that, as we employ a detailed architectural simulator (simulation is up to 1000x slower than native-execution), we scale the simulation memory size to reduce the warmup

gem5		SimpleSSD	
parameters	**value**	**parameters**	**values**
core	64-bit ARM, 8, 2GHz	**read/write/erase**	3us/100us/1ms
L1D$/L1I$	64KB, 64KB	**channel/package**	16/1
mem ctrler	1	**die/plane**	8/8
memory	DDR3, 2GB	**page size**	2KB
Kernel	4.9.30	**DMA/PCIe**	800MHz,3.0, x4
Image	Ubuntu 14.04	**DRAM cache**	1.5GB

Table 2: The configurations of gem5 and SimpleSSD.

App	Read Ratio	I/O size	I/Os per MegaInst.	Seq Ratio
bfs	0.997	238KB	0.025	0.70
gpgnu	1.000	134KB	1.985	0.78
gp	0.995	23KB	0.417	0.81
gzip	0.989	150KB	0.105	0.66
index	0.998	28KB	0.205	0.79
kmn	1.000	122KB	0.037	0.82
PR	0.995	30KB	0.367	0.71
ungzip	0.096	580KB	0.076	0.79
wcgnu	1.000	268KB	0.170	0.75
wc	0.999	25KB	0.548	0.89
ap	0.999	24KB	0.666	0.11
au	0.476	27KB	1.205	0.13
is	0.990	12KB	5.761	0.86

Table 3: The characteristics of workloads.

time of the CPU caches and DRAM. This is a common practice in architectural studies [1]. Our simulation environment integrates an accurate SSD simulator, SimpleSSD [21], which is attached to the computer system as a PCIe device. When booting, Linux running on gem5 recognizes SimpleSSD as a storage by creating a pair of NVMe SQ and NVMe CQ for each ARM core via the NVMe protocol [10]. Our SimpleSSD simulator is highly flexible and can configure various SSD prototypes. In this experiment, we configure SimpleSSD simulator as an ultra low latency SSD, similar to 800GB ZSSD [36]. The important characteristics of our gem5 simulation and SimpleSSD simulation setups are shown in Table 2.

Configurations. We implement four different computer systems by adding the optimization techniques proposed in FLASHSHARE, and compare them against Vanilla.

1. **Vanilla**: a vanilla Linux-based computer system running on ZSSD.
2. **CacheOpt**: compared to `Vanilla`, we optimize the cache layer of the SSD firmware by being aware of responsiveness.
3. **KernelOpt**: compared to `CacheOpt`, we further optimize the block I/O layer to enable latency-critical I/Os to bypass the software and hardware queues. In addition, this version also supports reordering between the NVMe driver and the NVMe controller.
4. **SelectISR**: compared to `KernelOpt`, we add the optimization of selective ISR (cf. Section 4).
5. **XLER**: based on `SelectISR`, we improve the I/O stack latency by employing our hardware accelerator.

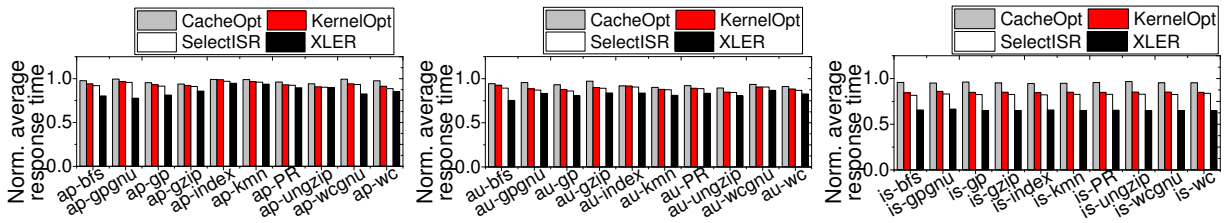
Workloads. We evaluate three representative online interactive workloads (latapp): *Apache (ap)*, *Apache-update (au)*, and *ImageServer (is)*. All these workloads create a web service scenario, in which a client thread is created to send client requests periodically and a server thread is created to receive the client requests. Once requests arrive in the server side, the server thread creates multiple worker threads to process these requests and respond to the client after the completion. For `Apache` and `Apache-update`, the requests are "SELECT" and "UPDATE" commands targeting a database, while the requests of `ImageServer` are image access requests. Note that the response time we measure in our experiments is the time between when the client thread issues a request and when the response of the request is received by the client. To satisfy the SLA requirements of the online applications, we select the request issue rate (as 400) right at the knee of the latency-load curve, which is also suggested by [28]. We also collect ten different offline workloads (thrapp) from BigDataBench (a Big Data benchmark Suite) [43] and GNU applications. The salient characteristics of our online and offline applications are listed in Table 3. In our evaluations, we co-run online interactive workloads and offline workloads together to simulate a real-world server environment.

In our evaluations, the response time means "end-to-end latency", collected from interacting workloads between client and server, which is different with other storage performance metrics that we used (i.e., I/O latency). Specifically, while the storage performance metrics only consider the characteristics of storage subsystems, the response time in our evaluations includes the request generate/send/receive latencies in a client, network latency, request receive/process/response latencies in a server, and storage-system latency.

5.2 Performance Analysis

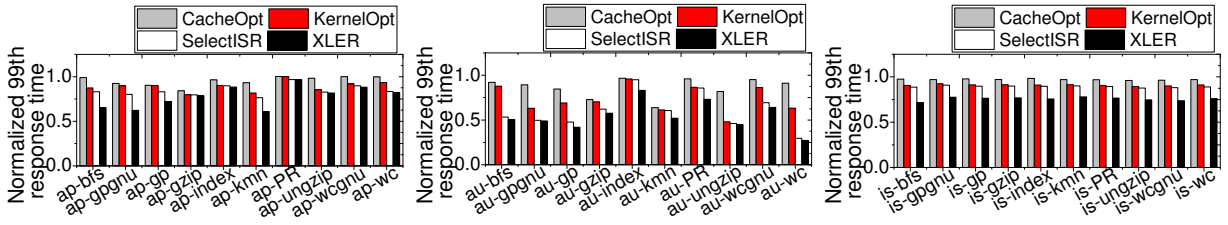
Figures 11 and 12 plot the average response time and the 99th response time, respectively, with the five different system configurations, normalized to those of `Vanilla`. Overall, `CacheOpt`, `KernelOpt`, `SelectISR` and `XLER` reduce the average response time by 5%, 11%, 12% and 22%, respectively, compared to `Vanilla`, while achieving 7%, 16%, 22% and 31% shorter 99th response times than `Vanilla` in that order.

`Vanilla` has the longest response time across all system configurations tested, because, it is not aware of the different workload attributes, and in turn loses the opportunity to optimize the kernel stack and flash firmware for latency-critical I/Os. In contrast, `CacheOpt` catches the



(a) `Apache (ap).` (b) `Apache-update (au).` (c) `ImageServer (is).`

Figure 11: Average response times of our online interactive applications normalized to `Vanilla`.



(a) `Apache (ap).` (b) `Apache-update (au).` (c) `ImageServer (is).`

Figure 12: 99th response times of our online interactive applications normalized to `Vanilla`.

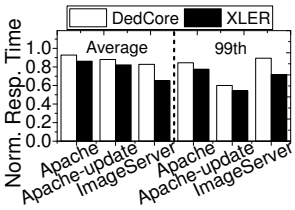


Figure 13: Response time analysis of using different polling techniques (normalized to `Vanilla`).

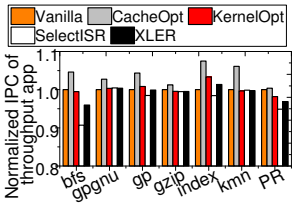
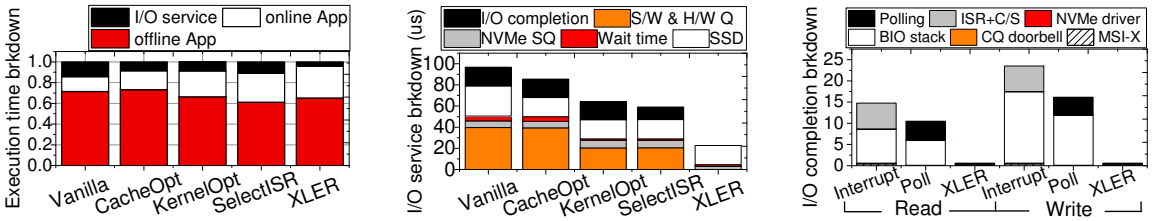


Figure 14: Performance analysis of offline applications (normalized to `Vanilla`).

workload attributes from the user processes and passes them to the underlying flash firmware. It further optimizes the SSD embedded cache by isolating latency-critical I/Os from the interference coming from throughput I/Os and customizing the cache access policy for latency-critical I/Os. As a result, it can accommodate more latency-critical I/Os in the SSD-embedded cache. As shown in Figures 11b and 12b, `CacheOpt` can reduce the average response time and 99th response time by up to 11% and 27%, respectively, if there are intensive write I/Os. Nonetheless, as flash firmware sits at the bottom of overall I/O stack, `CacheOpt` cannot effectively prevent throughput I/Os from impeding latency-critical I/Os from upper Linux kernel layers. Compared to `CacheOpt`, `KernelOpt` detects latency-critical I/O when it is inserted into the block I/O layer and creates a short path to send latency-critical I/O directly to the ULL SSD. Specifically, it enables latency-critical I/Os to directly bypass the software and hardware queues in the block I/O layer. It also collaborates with NVMe driver and NVMe controller to allocate an NVMe submission queue dedicated to latency-critical I/Os and fetch the latency-critical I/Os with a higher priority. Note that, `KernelOpt` can significantly reduce the 99th response time when offline applications generate intensive I/O requests (e.g., *ungzip*). The optimizations mentioned above can further reduce the average response time and the 99th response time by 6% and 8%, respectively, compared to `CacheOpt`. While `KernelOpt` works well for I/O submission optimization, it fails to handle the software overhead introduced by the interrupt-based I/O completion approach. For example, while an SSD read is as short as 3us, the ISR of the MSI-X interrupt and context switch collectively consume more than 6us. `SelectISR` selectively polls the latency-critical I/Os to avoid the use of ISR and context switch. Compared to the relatively long response time (i.e., more than 3 ms), the time saved from the ISR and context switch does not significantly reduce the average response time. However, in Figure 12b, we can observe that `SelectISR` can reduce the 99th request time by up to 34% in workload *bfs*. This is because, this compute-intensive workload creates multiple working threads that use up CPU resources and postpone the scheduling of the latency-critical I/Os. `SelectISR` secures CPU resources for the online interactive workloads as the CPU resources are not yielded to other user processes during polling. `XLER` can further reduce the average response time and 99th response time by 8% and 10%, respectively, compared to `SelectISR`. This is because, `XLER` simplifies the multiple queue management of the block I/O layer and NVMe driver, and accelerates the execution by employing customized circuits.

Since it would be possible to shorten the latency of storage stack by allocating a dedicated CPU core, we



(a) The execution time breakdown. (b) I/O stack breakdown. (c) I/O completion breakdown.

Figure 15: Execution time analysis of co-running `Apache-Index`.

also compare this alternative option with our hardware-assisted approach. Figure 13 shows the average response time and the 99th response time of online applications with a dedicated CPU core (`DedCore`) and a hardware accelerator (`XLER`). `XLER` reduces the average response time and the 99th response time by 12% and 15%, compared to `DedCore`, respectively. This is because, to leverage a dedicated core to poll NVMe CQs, `DedCore` requires intensive communications with the general cores, which are in process of the actual online applications. Unfortunately, such communications introduce different types of overheads associated with CPU cache flushes and spin-lock management.

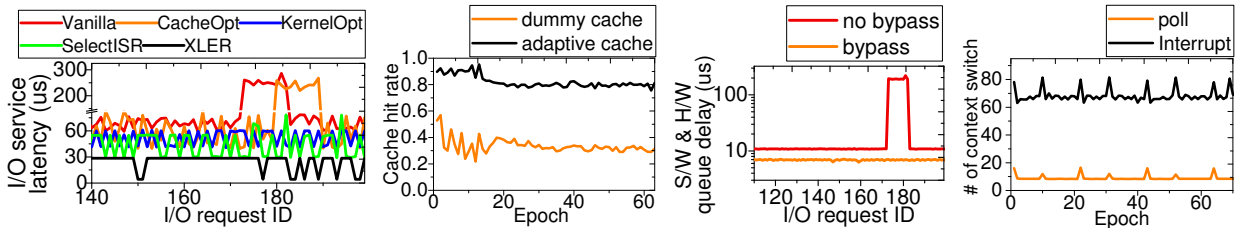
Although all our proposed techniques are oriented towards reducing the latency of the online interactive applications, offline workloads actually do not suffer from severe performance degradation. Figure 14 shows the performance of all evaluated offline workloads. Overall, FLASHSHARE does not degrade the performance of the offline applications, compared to `Vanilla` (the worst degradation observed is around 4%). This is because, the offline applications are not sensitive to the latency of each individual I/O, but instead rely on the storage bandwidth. Specifically, `CacheOpt` improves the performance of the offline applications by 3.6%, compared to `Vanilla`. This benefit comes mainly from the effective cache design and management. As `CacheOpt` separates the cache spaces for latency-critical I/O and throughput I/O, the throughput I/Os can better enjoy the fruits of short cache access latency without any competition originating from the latency-critical I/Os. On the other hand, we tune the delay time threshold and maximal number of throughput I/Os in the NVMe submission queue to make sure that all the delayed throughput I/Os are flushed by the NVMe controller before they start to introduce severe performance degradation. `SelectISR` degrades the performance of offline workloads by 2%, compared to `KernelOpt`. This is because, `SelectISR` uses up CPU resources for polling the latency-critical I/Os rather than executing the offline workloads. `XLER` achieves 1.2% higher performance than `SelectISR`, as it can effectively reduce the time spent for polling.

5.3 Effectiveness of Holistic Optimization

Figure 15a shows the execution time breakdown of co-running workloads *Apache* and *Index*. As shown in the figure, `CacheOpt` reduces the time needed to serve I/Os by 6%, compared to `Vanilla`, which in turn allows CPU to allocate more time for the offline application. On the other hand, `KernelOpt` postpones throughput I/Os, which blocks CPU from executing the offline application. For `SelectISR`, as CPU is used up for polling, less CPU time is allocated to the offline application. Finally, as `XLER` offloads the polling function to our hardware accelerator (cf. Section 4.3) and also reduces the time of I/O stack, both the online applications and offline applications can benefit from the reduced I/O service time.

Figure 15b plots the latency-critical I/O service breakdown between the co-running workloads, *Apache* and *Index*. `CacheOpt` reduces the average SSD access latency from 29 us to 18 us, compared to `Vanilla`, thanks to the short cache access latency. As the latency-critical I/Os are not queued in the software and hardware queues, the time for latency-critical I/O to pass through the block I/O layer is reduced from 39 us to 21 us when employing `KernelOpt`. Since the block I/O layer still needs to merge the latency-critical I/Os with the I/Os queued in software queues, the delay of the software and hardware queues cannot be fully eliminated. Compared to `KernelOpt`, `SelectISR` reduces the total I/O completion time by 5 us. We will present a deeper analysis of the I/O completion procedure shortly. As `XLER` removes the software and hardware queues in its implementation, it fully removes overheads of the block I/O layer.

Figure 15c plots the read and write I/O completion time breakdown. As shown in the figure, the polling function, interrupt service routine (ISR), context switch (CS) and block I/O layer collectively consume 96% of the I/O completion time, while the NVMe driver, sending MSI-X interrupt, and ringing CQ doorbell register together cost less than 0.5 us. Interestingly, although polling can remove the overhead caused by ISR and context switch, the polling function itself also introduces a 6 us delay. This delay is mainly caused by inquiring the



(a) Online I/O service. (b) Cache hit rates. (c) Blk-mq queue delay. (d) Context switch analysis.

Figure 16: Online interactive I/O execution time analysis.

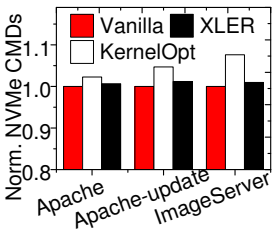


Figure 17: Analysis of # of NVMe commands.

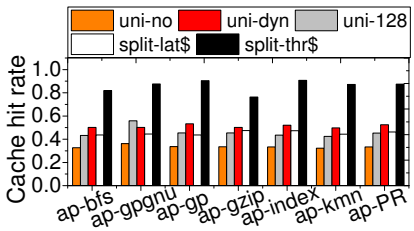


Figure 18: Various cache performance by co-running *Apache (ap)*.

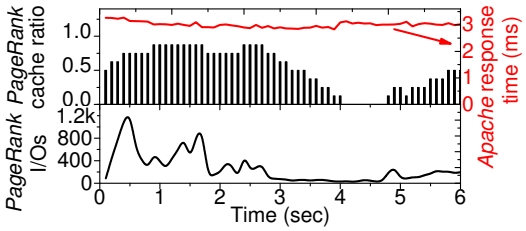


Figure 19: Analysis of performance dynamics when co-running *Apache* and *PageRank*.

IRQ locks of both BIO and NVMe CQ, setting current task status, and checking if there are any valid completion queue entries. In addition, although both the interrupt based approach and polling based approach execute the same block I/O completion function, polling reduces the average completion latency of block I/O by 4 us in both read I/Os and write I/Os. This is because, the interrupt-based approach context-switches CPU to other user processes which can pollute CPU caches, while the polling-based approach buffers the data used for I/O completion in the CPU cache.

5.4 I/O Stack Delay Analysis

Figure 16a shows the I/O service delay of the five different system configurations tested, for *Apache-Index* over time. In addition, Figure 16b plots the cache miss rates of a dummy cache (i.e., traditional SSD internal cache that has no knowledge of the host-side information) and our adaptive cache, while Figures 16c and 16d plot, respectively, the software and hardware queue latencies if we bypass or do not bypass latency-critical I/Os, and the number of context switches over time if we use the poll-based approach and the interrupt-based approach. `CacheOpt` exhibits a shorter I/O service delay than `Vanilla`, because it adjusts the cache space and prefetch granularity for the latency-critical I/Os and throughput I/Os, separately, resulting in fewer cache misses than the dummy cache in `Vanilla` (cf. Figure 16b). However, as shown in Figure 16a, `CacheOpt` cannot mitigate the long tail latency which is also observed in `Vanilla`, while `KernelOpt` successfully removes such tail latency. As shown in Figure 16c, the long tail latency comes from the software queue and hardware queue, if we buffer I/O requests in the software queue and hardware queue for merging and reordering. As `KernelOpt` enables the latency-critical I/Os to bypass the queues, it successfully eliminates the latency impact from the queues. `SelectISR` reduces the I/O service latency further, compared to `KernelOpt`. This is because, the polling-based approach can effectively reduce the number of interrupt service routine invocations and the number of context switches compared to the interrupt-based approach (cf. Figure 16d).

Since the I/O requests of online applications directly bypass the blk-mq layer under `KernelOpt`, the incoming I/O requests can lose their chances for merging, which can in turn increase the number of NVMe commands. Figure 17 shows the total number of NVMe commands, generated under different system configurations, namely `Vanilla`, `KernelOpt` and `XLER`. Compared to `Vanilla` that disables the bypassing scheme, `KernelOpt` increases the number of NVMe commands by only 2%. This is because the latency-critical I/O requests (coming from the online applications) exhibit low locality, and their arrival times are sporadic. Thanks to its merge logic, `XLER` further reduces the number of NVMe commands by 0.4%, on average, compared to `Vanilla`.

5.5 Sensitivity to Embedded Cache Design

Figure 18 gives the cache hit rate of various cache configurations when co-running *Apache* with offline workloads. Specifically, *uni-no*, *uni-128* and *uni-dyn* con-

figure a uniform cache for both the latency-critical and throughput I/Os. However, *uni-no* disables page prefetching and *uni-128* always prefetches 128 pages when cache misses, while *uni-dynfetch* employs the adaptive prefetching scheme we proposed (cf. Section 4.2). On the other hand, *split-lat$* and *split-thr$* represent the separate caches owned by the latency-critical I/Os and throughput I/Os, respectively. The separate cache employs adaptive prefetch scheme and adjusts cache space at runtime. *Apache* is a representative workload which randomly accesses the storage (cf. Table 3). As shown in the figure, although offline applications access the storage in a sequential manner, *uni-128* achieves only a 12% higher cache hit rate than *uni-no*. This is because, the random access pattern exhibited by *Apache* can pollute the cache space and make prefetching less effective. On the other hand, *uni-dyn* adjusts the prefetch granularity to small number of pages at runtime so that prefetching pages for latency-critical I/Os will not pollute all the caches. *split-lat$* does not achieve a high cache hit rate, due to the random access pattern of *Apache*. However, as we split the cache and isolate the interference from online applications, *split-thr$* achieves a great improvement in terms of hit rates.

Figure 19 further demonstrates the effectiveness of our dynamic cache partitioning scheme. Specifically, the lower half of Figure 19 shows the number of I/O requests, generated by offline applications during the execution, while the upper half shows the dynamics of the cache spaces, allocated to the offline application (in terms of ratio), and in parallel, demonstrates the response time of the online application. When there is an I/O burst imposed by *PageRank* (0∼2.8 seconds), our SSD controller isolates the negative impact of this I/O burst by preserving the cache spaces for *Apache* as 23% of the total cache space, on average. By partitioning the cache space being aware of the responsiveness for different applications, our cache partitioning secures just enough cache spaces for both *PageRank* and *Apache* such that the response time of *Apache* can be sustained at 3.1 ms while *PageRank* improves the cache hit rates by approximately 36% compared to a dummy cache. In cases where the offline application requests I/O services less than before (3∼6 seconds), our dynamic cache partitioning method increases the faction of internal cache spaces for the online application, which can be used for the application to perform pre-fetch or read-ahead in helping with an immediate response from the internal cache.

6 Related Work

In various computing domains, there are multiple studies to vertically-optimize storage stack [16, 24, 47, 50, 19, 22]. For example, [19] and [22] take flash firmware out of an SSD and locate it to the host, in order to remove the redundant address translation between a file system and FTL. In comparison, [47] proposes multiple partitioned caches on the host side. These caches understand multiple client characteristics by profiling the hints passed from one or more clients through out-of-bound protocol. However, the application hints are used only for cache management; such hints/approaches have no knowledge of the underlying storage stack and they do not consider the potential benefits of ULL. [16] optimizes mobile systems from the viewpoint of a file system and a block I/O device to improve the performance of databases such as SQLite. However, this optimization is applied only for the logging performance of mobile databases; it cannot be applied to other applications and cannot expose ULL to them. [50] schedules write requests by considering multiple layers on the kernel-side. While this can improve the write performance, such writes can block reads or ULL operations at the device level, as the ULL SSD also includes embedded DRAM caches and schedulers.

[24] observes that there exists an I/O dependency between background and foreground tasks. This dependency degrades overall system performance with a conventional storage stack since kernel always assigns a high priority to I/O services generated from the foreground tasks and postpones the background I/O requests. This I/O stack optimization allows the foreground tasks to donate their I/O priority to the background I/O services, when an I/O dependency is detected. However, this approach does not well fit with I/O workloads that often exhibit no I/O dependency. In particular, multiple applications executed on a same server (for a high resource utilization and energy efficiency) are already independent (as they operate in a different domain).

[32] and [23] propose sharing a hint with the underlying components to have a better data allocation in disk array or virtual machine domains. Similarly, [51] modifies a disk scheduler to priotize I/O requests, which are tagged by interactive applications. While most prior approaches leverage the hints from users/applications to improve the design of specific software layers, they do not consider the impact from the other parts of the storage stack.

[41] simplifies the handshaking processes of the NVMe protocol by removing doorbell registers and completion signals. Instead of using MSI, it employs a polling-like scheme for the target storage system. More

recently, [16, 7, 40, 3, 48] also observed that polling can consume significant CPU cycles to perform I/O completion. [7, 44] applies a hybrid polling method, which puts the core into sleep for a certain period of time, and just performs poll the request only when the sleep time has passed. While this strategy can reduce the CPU overheads to some extent, it is not trivial to determine the optimal time-out period for sleeps. Even in cases where system architects can decide a suitable time-out period, I/O request patterns can dynamically change and the determined time-out period may not be able to satisfy all user demands. Further, this can waste CPU cycles (if the time-out is short) or make the latency of I/O request longer (if for example the time-out is longer than the actual completion time). In addition, unlike our FLASHSHARE, the hybrid scheme cannot reduce the latency burden imposed by the software modules in the storage stack.

7 Discussion and Future Work

While the hardware accelerator of FLASHSHARE can perform a series of time-consuming tasks such as NVMe queue/entry handling and I/O merging operations on be half of CPUs, the accelerator employed in the target system is optional; we can drop the accelerator in favor of a software-only approach. This software-only FLASHSHARE (as a less-invasive option) makes performance of the server-side storage system approximately 10% worse and consumes 57% more storage-stack side CPU-cycles than hardware-assisted FLASHSHARE. Note that the hardware accelerator does not require high-performance embedded-cores and needs no high-capacity memory either, since it only deals with NVMe-commands and reuses the system-memory for data-management (via PRPs).

Bypassing a request is not a new idea, but it requires the proposed optimization of FLASHSHARE to apply such bypassing concept from the kernel to the firwmare. For example, bypassing scheme has been well investigated to improve the throughput of network [27]. While network kernel bypassing transfers data by directly mapping user memory to physical memory, the storage stack cannot simply adopt the same idea, due to ACID capability supports and block interface requirements. In addition, bypassing in block interface devices should still go through filesystem, page-cache, scheduler and interface-driver for user-level services. This introduces higher complexity and multiple interface boundaries than network, and also renders the direct mapping between user memory and phyiscal memory not a viable option. On the other hand, SPDK [39] is designed for a specific-purpose, namely, NVMe-over-Fabric that requires client-side file-systems or RocksDB-based applications, which is different from the datacenter's co-located workload scenario that FLASHSHARE works on.

Even though FLASHSHARE can remove a significant chunk of CPU-side overheads (around 79%, compared to naive-polling) with 20%$\sim$31% better user experience from the client-side, it also has a limit; FLASHSHARE is mainly designed towards accelerating the services in the cluster servers, but it unfortunately does not fit for the workload scenarios that rent computing-capability to multiple tenants, such as Infrastructure as a Service (IaaS). In our on-going work, we are extending FLASHSHARE with a different type of storage, such as multi-streamed (or ported) SSDs [52, 25, 49, 35] over diverse storage I/O virtualization techniques.

8 Acknowledgement

The authors thank Steven Swanson of UCSD for shepherding their paper. This research is mainly supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, Yonsei Future Research Grant (2017-22-0105) and MemRay grant (2015-11-1731). The authors thank Samsungs Jaeheon Jeong, Jongyoul Lee, Se-Jeong Jang and JooYoung Hwang for their SSD sample donations. N.S. Kim is supported in part by grants from NSF CNS-1557244 and CNS-1705047. M. Kandemir is supported in part by grants by NSF grants 1822923, 1439021, 1629915, 1626251, 1629129, 1763681, 1526750 and 1439057. Myoungsoo Jung is the corresponding author.

9 Conclusion

We propose FLASHSHARE, which punches through the storage stack from kernel to firmware, helping ULL SSDs satisfy different levels of user requirements. At the kernel level, we extend the data structures of the storage stack to pass attributes of (co-running) applications through all software modules of the underlying OS and device. Given such attributes, the block layer and NVMe driver of FLASHSHARE custom-manage the I/O scheduler and interrupt handler of NVMe. The target ULL SSD dynamically partitions the internal DRAM and adjust its caching strategies to meet diverse user demands. By taking full advantage of the ULL services, this holistic approach significantly reduces the inter-application I/O interferences in servers co-running multiple applications, without modifying any of the applications.

References

[1] ALIAN, M., ABULILA, A. H., JINDAL, L., KIM, D., AND KIM, N. S. Ncap: Network-driven, packet context-aware power management for client-server architecture. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 25–36.

[2] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[3] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), IEEE Computer Society, pp. 385–395.

[4] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing* (2007), ACM, pp. 1126–1130.

[5] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 3*, 4 (2004), 837–863.

[6] CHEONG, W., YOON, C., WOO, S., HAN, K., KIM, D., LEE, C., CHOI, Y., KIM, S., KANG, D., YU, G., ET AL. A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time. In *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International* (2018), IEEE, pp. 338–340.

[7] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, p. 42.

[8] FARRINGTON, N., AND ANDREYEV, A. Facebook's data center network architecture. In *Optical Interconnects Conference, 2013 IEEE* (2013), Citeseer, pp. 49–50.

[9] FREE SOFTWARE FOUNDATION. nice. `http://www.gnu.org/software/coreutils/manual/coreutils.html#nice-invocation`.

[10] GOUK, D., KWON, M., ZHANG, J., KOH, S., CHOI, W., KIM, N. S., KANDEMIR, M., AND JUNG, M. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *Proceedings of the 2018 51rd Annual IEEE/ACM International Symposium on Microarchitecture* (2018), IEEE Computer Society.

[11] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), IEEE, pp. 24–33.

[12] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The harey tortoise: Managing heterogeneous write performance in ssds. In *USENIX Annual Technical Conference* (2013), pp. 79–90.

[13] INTEL CORPORATION. Intel 535. `https://ark.intel.com/products/86734/Intel-SSD-535-Series-240GB-2_5in-SATA-6Gbs-16nm-MLC`, 2015.

[14] INTEL CORPORATION. Intel 750. `https://ark.intel.com/products/86740/Intel-SSD-750-Series-400GB-12-Height-PCIe-3_0-20nm-MLC`, 2015.

[15] INTEL CORPORATION. Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2016.

[16] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/o stack optimization for smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 309–320.

[17] JIN, Y., WEN, Y., AND CHEN, Q. Energy efficiency and server virtualization in data centers: An empirical investigation. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on* (2012), IEEE, pp. 133–138.

[18] JUNG, M. Exploring parallel data access methods in emerging non-volatile memory systems. *IEEE Transactions on Parallel & Distributed Systems*, 3 (2017), 746–759.

[19] JUNG, M., AND KANDEMIR, M. Middleware - firmware cooperation for high-speed solid state drives. In *Middleware '12* (2012).

[20] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review* (2013), vol. 41, ACM, pp. 203–216.

[21] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. Simplessd: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters PP*, 99 (2017), 1–1.

[22] JUNG, M.-S., IK PARK, C., AND OH, S.-J. Cooperative memory management. In *US 8745309* (2009).

[23] KIM, J., LEE, D., AND NOH, S. H. Towards slo complying ssds through ops isolation. In *FAST* (2015), pp. 183–189.

[24] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the i/o path: A holistic approach for application performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2017), FAST'17, USENIX Association, pp. 345–358.

[25] KIM, T., HAHN, S. S., LEE, S., HWANG, J., LEE, J., AND KIM, J. Pcstream: automatic stream allocation using program contexts. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018), USENIX Association.

[26] LEVERICH, J., AND KOZYRAKIS, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 4:1–4:14.

[27] LIU, J., WU, J., AND PANDA, D. K. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming 32*, 3 (2004), 167–198.

[28] LO, D., CHENG, L., GOVINDARAJU, R., BARROSO, L. A., AND KOZYRAKIS, C. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 301–312.

[29] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 450–462.

[30] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 248–259.

[31] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *FAST* (2003), vol. 3, pp. 115–130.

[32] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 57–70.

[33] MURUGAN, M., AND DU, D. H. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on* (2011), IEEE, pp. 1–12.

[34] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. *Informed prefetching and caching*, vol. 29. ACM, 1995.

[35] RHO, E., JOSHI, K., SHIN, S.-U., SHETTY, N. J., HWANG, J.-Y., CHO, S., LEE, D. D., AND JEONG, J. Fstream: managing flash streams in the file system. In *16th USENIX Conference on File and Storage Technologies* (2018), p. 257.

[36] SAMSUNG MEMORY SOLUTIONS LAB. Ultra-low latency with samsung z-nand ssd. `https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf`, 2017.

[37] SHIM, H., SEO, B.-K., KIM, J.-S., AND MAENG, S. An adaptive partitioning scheme for dram-based cache in solid state drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–12.

[38] SMITH, K. Garbage collection. *SandForce, Flash Memory Summit, Santa Clara, CA* (2011), 1–9.

[39] SOURCE, I. O. org. storage performance development kit (spdk), 2016.

[40] SUNGJOON KOH, CHANGRIM LEE, M. K. M. J. Exploring system challenges of ultra-low latency solid state drives. In *The 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2018), IEEE.

[41] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., LE MOAL, D., BUNKER, T., XU, J., SWANSON, S., ET AL. Dc express: shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX conference on File and Storage Technologies* (2014), USENIX Association, pp. 309–315.

[42] WANG, J., PARK, D., PAPAKONSTANTINOU, Y., AND SWANSON, S. Ssd in-storage computing for search engines. *IEEE Transactions on Computers* (2016).

[43] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ZHENG, C., LU, G., ZHAN, K., LI, X., AND QIU, B. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2014), pp. 488–499.

[44] WEST DIGIT. I/o latency optimization with polling. `https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final_0.pdf`.

[45] WIKI. Cpu cache. `https://en.wikipedia.org/wiki/CPU_cache`.

[46] WORKGROUP, N. E. Nvm express revision 1.3. `https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf`.

[47] YADGAR, G., FACTOR, M., LI, K., AND SCHUSTER, A. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems (TOCS) 29*, 2 (2011), 5.

[48] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *FAST* (2012), vol. 12, pp. 3–3.

[49] YANG, J., PANDURANGAN, R., CHOI, C., AND BALAKRISHNAN, V. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), ACM, p. 3.

[50] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 474–489.

[51] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First class support for interactivity in commodity operating systems. In *OSDI* (2008), vol. 8, pp. 73–86.

[52] YONG, H., JEONG, K., LEE, J., AND KIM, J.-S. vstream: virtual stream management for multi-streamed ssds. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018), USENIX Association.

Orca: Differential Bug Localization in Large-Scale Services

Ranjita Bhagwan Rahul Kumar Chandra Sekhar Maddila Adithya Abraham Philip

Microsoft Research India

Abstract

Today, we depend on numerous large-scale services for basic operations such as email. These services are complex and extremely dynamic as developers continuously commit code and introduce new features, fixes and, consequently, new bugs. Hundreds of commits may enter deployment simultaneously. Therefore one of the most time-critical, yet complex tasks towards mitigating service disruption is to localize the bug to the right commit.

This paper presents the concept of *differential bug localization* that uses a combination of differential code analysis and software provenance tracking to effectively pin-point buggy commits. We have built Orca, a customized code search-engine that implements differential bug localization. On-Call Engineers (OCEs) of Orion, a large enterprise email and collaboration service, use Orca to localize bugs to the appropriate buggy commits. Our evaluation shows that Orca correctly localizes 77% of bugs for which it has been used. We also show that it causes a 3x reduction in the work done by the OCE.

1 Introduction

Orion [1] is a large enterprise email and collaboration service that supports several millions of users, runs across hundreds of thousands of machines, and serves millions of requests per second. Thousands of developers contribute code to it at the rate of hundreds of commits per day. Dozens of new builds are deployed every week. Software bugs are bound to be common in such a complex and dynamic environment. It is critical to detect and promptly localize such bugs since service disruptions lead to customer dissatisfaction and significantly lower revenues [21].

When a service disruption happens because of a software bug, the first-step towards mitigating its effect is to localize the responsible bug to the right commit. We call this *commit-level bug localization*. This is a non-trivial task since the intense pace of development demands that multiple commits be aggregated into a single deployment. In addition, commit-level bug localization needs to happen as quickly as possible so that buggy commits can be reverted promptly thereby restoring service health. About half of all Orion's service disruptions are caused by software bugs.

Unfortunately, bug localization in large services such as Orion is a cumbersome, time-consuming, and error-prone task. The *On-Call Engineers (OCEs)* are the first points-of-contact when a disruption occurs, and they are responsible for bug localization. Though knowledgeable, on-call engineers can hardly be expected to have complete and in-depth understanding of all recent commits. Moreover, bugs that emerge after deployment are complex and often non-deterministic. And yet, very few tools exist to enable OCEs to perform this critical task.

Our goal is to build a tool that will help OCEs correctly and swiftly localize a bug to the buggy commit. Over a period of eight months, we studied post-deployment bugs, their symptoms, the buggy commits that caused them, and the current approach to bug localization that Orion's OCEs follow. We made four key observations.

1) *Bug localization is time-critical, bug fixing is not.* When a bug disrupts a service, the OCE's task is to keep the service disruption time to a minimum. She finds the buggy commit as fast as possible and reverts it rather than wait for the concerned developer to fix the bug. The reason is that, depending on the complexity of the bug, the developer may take a long time to fix it. Therefore to keep disruption to a minimum, it is better to revert the buggy commit first and introduce the fix at a later time. Thus, fast commit-level bug localization is critical.

2) *Rich monitoring infrastructure exists but is insufficient because of uncaptured dependencies.* Since service disruptions are a major concern, developers have created thousands of active *probes* that periodically monitor service-components or API calls and raise *alerts* if they fail. Despite this, bug localization is a challenge because a probe to a component may fail not because of any change to the component itself, but because of a change to another component that depends upon it. For instance, a server-side probe failed with an exception `Type RecipientId not supported` because a developer made a commit to client-side code that added support for the data type `RecipientId` without adding support on the server-side. To make matters worse, as the service evolves fast, new dependencies emerge at a rapid

[1] Name changed.

rate and no tool can completely capture all of them.

3) *Symptom descriptions and their causes tend to have similar terms.* We have found that when a probe detects a bug, a similarity often exists between terms in the unhealthy probe name or exception text that it generates and the source-code change that caused the bug. In the example mentioned in the previous paragraph, the term `recipient` occurs in both symptom (the exception text) and cause (added support to the data type on the client). We also see this similarity in some customer complaints as well which predominantly use natural language. For instance, a customer recently complained that "*Email ID suggestions for people I know is not working.*". The cause for this was an incorrect change to a function named `PeopleSuggest`.

4) *Bugs may appear well after the buggy commit is deployed.* We observed that while the symptoms of a bug appear in a current build, the cause may be a commit deployed in a much older build. These are particularly challenging for the OCE to localize because they have to investigate, in the worst-case, all commits in the current build before moving on to investigate an older build. This can significantly lengthen service disruptions.

Keeping these observations in mind, we design a novel search technique that we call *differential bug localization*. Using descriptions of the bug as a query, we detect changes to the abstract syntax tree in the source-code and search only these changes for text-based similarity. We call this *differential code analysis*. To find offending commits in older builds, we introduce a construct called the *build provenance graph* that captures dependencies between builds. We designed Orca, a custom code search-engine that leverages differential bug localization to provide a ranked list of "suspect" commits.

The Orion service has integrated Orca into its alerting and monitoring processes. This paper describes Orca and makes the following contributions:

- We provide a study of post-deployment bugs found in the Orion service and their characteristics (Section 3).
- We introduce *differential bug localization*, which uses two constructs: differential code analysis of the abstract syntax tree, and the build provenance graph. In addition, we use a prediction of commit risk to call out riskier commits in the list of potential root-causes (Section 4).
- We have designed Orca, a tool that Orion's OCEs are actively using to localize bugs (Section 5).
- We provide an evaluation of Orca for bugs found in the Orion service (Section 6).

To the best of our knowledge, ours is the first study of a bug localization tool deployed on a large-scale enterprise service. We have evaluated Orca on 48 post-deployment bugs found in Orion since October 2017. We show that Orca correctly localizes 37 out of 48 bugs for a recall of 77%. In 30 of the 37 cases, the correct commit was ranked in the top 5 records shown by our UI (Section 6). We also show that Orca causes a $3\times$ reduction in the work done by the OCE.

We have designed Orca for usability and ease-of-adoption. While this paper concentrates on Orion's deployment of the tool, Orca has been deployed on five other services within our enterprise. Our techniques are generic and extend well to other large services. To make it easy for OCEs to use the tool interactively, we have optimized its performance through multiple caching and preprocessing techniques. Our user-interface provides results with an average run-time of 5.9 seconds per query.

2 Related Work

The Programming Languages, Software Engineering and Systems communities have extensively studied bug detection, bug localization and debugging. While Orca takes inspiration from some of this prior work, it targets a fundamentally different application space, i.e. large-scale service deployments. Also, Orca is meant to be used by on-call engineers, not developers.

A bug localization tool for such services needs to be *fast*: the query response time should be at most a few seconds since OCEs will use the UI interactively. It should be *general*: the techniques should support code in different languages and should not need an OCE or developer to provide specification. It should be *non-intrusive*: we should not require any changes to the service's existing coding and deployment practices. Finally, it should be *adaptive:* it should work in an extremely dynamic and changing environment. We now describe prior work in the area and explain why it does not satisfy some or all of these requirements.

IR-based Bug localization techniques [3, 25, 35, 36, 38, 42, 49] use a given bug-report to search, based on textual similarity, for similar bug reports in the past. For each match, they localize the bug at the *file-level*, not at commit-level, to files that have been changed to address similar bug-reports. Wang et al. [38] present a structured and detailed study of the various techniques that are used for information-retrieval based bug localization. They use similar search techniques over five major concepts: version history, bug reports, stack traces, and reporter information.

While these techniques are fast, general and non-intrusive, they assume a stationary system, i.e. they as-

sume that there is an inherent similarity between the current version of the system and previous versions. This is fundamentally not true in service deployments. For instance, Orion experiences a change in module dependencies at the rate of 1% to 3% every month, and some source-code files consistently change more than once a day.

Moreover, prior work has mostly studied software products, not deployed services. Comparatively, the work presented in this paper is different because it focuses on (a) dynamic deployments of software services, and (b) both structured and unstructured queries for localization of the manifested issue. Finally Orca is deployed and being used in real-time on a large service deployment. To our knowledge, existing bug localization techniques have not seen such scale of deployment.

Dynamic Analysis techniques are the most commonly used and widely studied approaches for detecting bugs and issues in software. Testing and automated fuzz testing techniques [6, 18] provide an effective method to automatically generate test-cases that produce random inputs for the underlying software. Although these techniques are useful, they are complementary to our approach. Testing is never complete and does not provide guarantees about the correctness of the software. As a result, in spite of comprehensive testing, bugs still emerge regularly in service deployments, as we have noticed with Orion.

To find post-deployment bugs, previous work has presented statistical techniques [7, 9, 26] to automatically isolate bugs in programs by tracking and analyzing *successful* and *failed* runs of the program. While these technique hold promise, they are intrusive: they requires fine-grained instrumentation and a large number of program traces from the service deployment. Given the stringent performance needs and dynamism in services, we do not have the luxury of utilizing such techniques.

Delta Debugging techniques [12, 45] help automate the problem of debugging by providing the debugger or programmer information about the state of the program in passing and failing runs. The possible search space of the root cause is systematically pruned by using information from the various runs and by creating new executions. The ideas in delta debugging rely heavily on *program slicing* [1, 39]. Given its requirements, delta debugging tends to require a large number of tests and the data from instrumented programs. Hence, it is intrusive.

The GIT `bisect` command [17] is similar in flavor. Given the last good commit, it uses an algorithm based on binary search to go through subsequent commits, repeatedly test the code, and ultimately localize the problem to the buggy commit. However, like delta debugging, this may require a large number of testing cycles. Moreover, a bug in a large deployment may not be reproducible by simple testing.

Static analysis entails analyzing software without executing the programs in question. The analysis may be performed at the level of the source-code itself, or at the level of the object code that is generated for execution (byte code or binary). Analysis is performed by automated tools that tend to be rooted in some formal method such as *model checking* [11], *symbolic execution* [29], and *abstract interpretation* [13]. Although such techniques have shown significant promise in the past, performing such analysis on a large scale for services has proven to be very slow and intractable. Performing program analysis and verification at smaller scales for individual components is the current limit of such techniques.

Tools such as Semmle [14] provides a unified framework that implements various program analysis techniques and correctness checks. In our experience, Semmle has proven to be somewhat useful for simple situations, but it lacks generality.

Differential static analysis techniques such as SymDiff [24] are immediately relevant to the problem discussed in this paper. But differential analysis techniques are usually *property driven*; two versions of the program are analyzed with respect to a specific correctness property. For example, the analysis may be performed for asserting differences in the new version w.r.t. null pointer dereferences. We believe this approach too lacks generality since it is not feasible to enumerate all such properties of code in large dynamic services with multiple dependent components. Yet, we do draw inspiration from this work to build Orca's AST-based differential analysis.

Tools such as Gumtree [15] use differences in ASTs to derive accurate edit scripts. Their techniques need to be fine-grained and therefore use heavyweight algorithms that implement isomorphism detection. Orca uses differences in ASTs with a different goal: for a changed file, we use the AST difference to enumerate all changed entities, such as changed variables, methods, classes, namespaces, etc. So we use faster, more coarse-grained heuristics than Gumtree, making our techniques much more performant.

Log Enhancement techniques [44, 47] improve log-based debugging by making logs richer and more targeted towards diagnosability. We believe such work is complementary to our approach and Orca can gain significantly with such techniques.

3 Overview

In this Section, we first describe the system development lifecycle of the Orion enterprise email service and the role of the OCE. We next describe the characteristics

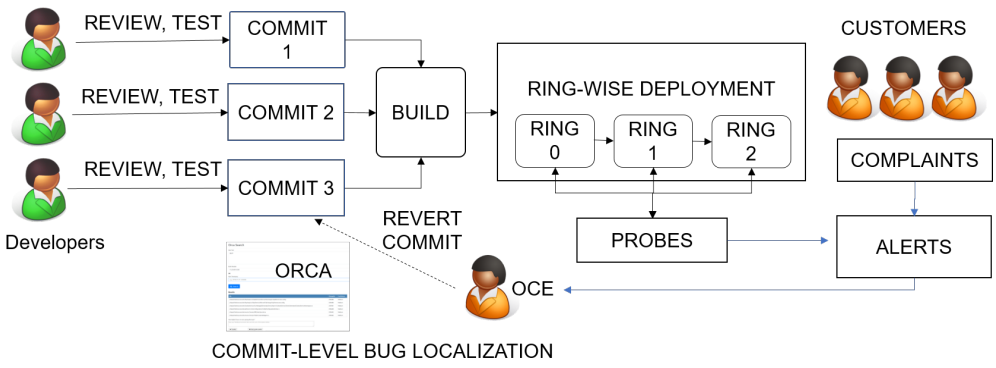


Figure 1: System Development Lifecycle (SDLC) of the Orion Service.

of post-deployment bugs in Orion and motivate the approaches we adopt in Orca. We provide an overview of Orca and its goals, and finally state Orca's scope.

3.1 System Development Lifecycle

Figure 1 shows Orion's system development lifecycle (SDLC) and Table 1 holds a summary definition of each term we use for the reader's convenience. Multiple developers *commit* code, where a commit varies in complexity from a small tweak to a single file to changes to hundreds, even thousands of files. These commits are *reviewed* by one or more reviewers that the developer chooses. After multiple iterations with the reviewers, the commits are tested using unit, integration, and functional tests. Periodically, the administrator creates a new *build* by combining a set of commits. A build is a unit of deployment of the service and may contain just one, or hundreds of commits.

Builds are deployed in stages onto *rings*. A ring is a pre-determined set of machines on which the same build runs. The build is first deployed onto the smallest ring, or "Ring 0", consisting of a few thousand machines. When it is considered safe, the build progresses through multiple rings such as Ring 1 and Ring 2 until it is finally deployed world-wide. The idea of this staged deployment is to find bugs early in the life-cycle.

Once the build is deployed to a ring, several tools monitor it. A tool may use passive or active monitoring techniques, either analyzing logs or sending periodic *probes* to a component. It uses anomaly detection techniques to raise an *alert* that the OCE receives.

If only the machines running a specific build raise alerts, the OCE concludes that the build is buggy and she begins bug localization. Roughly half of all alerts in Orion are caused by bugs. Consequently, bugs are a significant reason for service disruption. If the alerts are not confined to a particular build, the bug is likely due to other reasons such as faulty networks or hardware misconfiguration. Root-causing infrastructure issues is not our focus as several tools already exist for this purpose [43, 46].

To localize the bug to a commit, the OCE picks the commit that she feels is most likely to cause an issue and contacts the developer who created it. If the developer responds in the affirmative that her commit may indeed have caused the bug, the commit is immediately reverted and the service is restored to a healthy state. Note that the developer does not necessarily debug or fix the bug before responding to the OCE. If the developer says that their commit is not responsible for the bug, then the OCE picks the next most likely commit, and repeats the process until the service becomes healthy. Compared to a novice, an experienced OCE with domain-knowledge may pick the correct commits more promptly and therefore restore the service much faster. Orca removes this dependency on experience and domain-knowledge by codifying it in its search algorithms.

3.2 Post-Deployment Bugs

Over a period of eight months, we analyzed various post-deployment bugs and the buggy source-code that caused them. Table 2 outlines a few characteristic issues. The table shows the type of alert, it provides an overview of the symptom, and a description of the root-cause. It also shows the number of commits (and the number of files) that an OCE has to consider while performing bug attribution which, in some cases is more than 200.

In general, we have found that bugs fall predominantly into one of the following categories:

- **Bugs specific to certain environments.** Often, a component starts failing because files implementing that component have a bug (Bugs 2, 5). Usually, the failures happen only for a specific type of client such as web-based clients, or in a specific region such as Japan. Tests do not catch the bug since

Term	Definition
Commit	Set of file changes made by one developer.
Review	Recommendations made by one or more developers for a commit.
Build	Unit of deployment for the service consisting of one or more commits.
Ring	Set of machines onto which a build is deployed.
Probe	Periodic checks to functions/APIs to ensure they are working as expected.
Alert	An email- or web-based notification that warns the OCE of a problem.

Table 1: Terms used in the SDLC description and their definitions

not all configurations, clients and environments are tested.

- **Bugs due to uncaptured dependencies.** Dependencies can be of various types. In Bug 10, a server-side implementation is modified without appropriately modifying the client-side code. This happens because developers often overlook such dependencies as no compile-time tool captures them completely. Another example of an uncaptured dependency is Bug 4. A commit modifies a certain library, but unbeknownst to the developer, another component depends on certain features in the older version of the library and stops working correctly.

- **Bugs that introduce performance overheads.** Several probes track performance issues. For instance, in Bug 8, a code addition that was not thread-safe caused CPU overload that slowed down the service. The bug emerges only when a large number of users use the service. Hence it is not caught in testing.

- **Bugs in the user-interface.** A UI Feature starts misbehaving, so a customer complains. An example of this is Bug 1.

3.3 Orca Overview

Studying these bugs and observing the OCEs gave us valuable insights. We state these insights, and describe how Orca 's design is influenced by them.

Often, the same meaningful terms occur both in the symptom and the cause. Table 2 captures this under the "Term Similarity" column. Some matched terms are proper nouns such as the component name (Bug 3) or global data types (Bug 7). They can also be commonly used terms such as protocols (`Imap` in Bug 9) or the function performed (`suggest` in Bug 1 and `migrat` in Bug 6). Given the term similarity between symptom and cause, we designed Orca as a custom-designed search engine, using the symptom as the query-text, and giving us a ranked list of commits as results. Orca searches for the symptomatic terms in names of the modified code by performing differential code analysis on the abstract syntax tree. We describe this procedure in Section 4.1.

Testing and anomaly detection algorithms do not always find a bug immediately. A bug may start surfacing in a new build despite being introduced through a commit to a much older build. Bug 3 in Table 2 is an example. We introduce a *build provenance graph* to allow Orca to expand its search to older builds from which the current build has been derived. We describe this in Section 4.2.

Builds may have hundreds of commits, so manually attributing bugs can be long-drawn task. For instance, Bug 2 appears in a build that had 201 commits. Bug 3 appears in a build with 160 commits but the root-cause was in the previous build which had 41 commits. The OCE is faced with the uphill task of analyzing, in these cases, up to 200 commits before discovering the buggy commit. OCEs often work at odd hours and are constantly pressed for time. Orca therefore ranks commits based on a *prediction of commit risk.* Orca uses machine-learning and several features such as developer experience, code hot-spots and commit type to make this prediction. We describe this in Step 4 of Section 4.3.

To facilitate its use, we have built an Orca user interface and leverage caching and parallelism to ensure an interactive experience for the OCE. We describe our optimizations in Section 5.

There are thousands of probes in the system, and probe failures and exceptions are continuously logged. Therefore there is rich data on what symptoms, or potential queries to Orca look like. This allows us to track frequency of terms that appear in the queries and use the *Inverse Query Frequency (IQF)* rather than the Inverse Document Frequency (IDF) in our search rankings. We explain this further in in Section 4.3.

3.4 Orca Scope

In this section, we elaborate what Orca does *not* aim to solve. This is primarily because existing techniques already address these issues.

Orca does not target issues caused by faults in the infrastructure. Several techniques [22, 43] exist to do this. Orca does not solve the anomaly detection problem directly as several techniques already exist for this [4, 10].

No.	Type	Symptom	Cause	Term Similarity	Commits
1	Customer complaint	"People Suggestion" feature, that suggests potential recipients for an email, was not working for a subset of users.	A "people ranking" algorithm was incorrectly modified.	A variable used the keyword `suggest` in the modified function.	33
2	Probe	An email synchronization problem was detected.	A buggy commit to the synchronization component caused requests coming only from web-based clients to fail.	The probe contained the name of the synchronization component, which was also in the directory path of the modified files.	201
3	Probe	A worker process for a specific component started crashing repeatedly.	Incorrect configuration changes to the component's environment caused this. The commit was to a previous build but bug showed later only after a large number of users hit it.	The component name matched in the class name of modified code.	201
4	Customer complaint	Authentication process for some applications that used REST started crashing.	A library that these applications depended upon was modified but was not tested for all applications.	Keyword `auth` was in the pathname of the change.	46
5	Probe	Threads were getting blocked in processing, large delays were noticed in REST calls made by a web service.	HTTP client code for the service had been modified to make some synchronous calls asynchronous.	The component name matched a modified user-agent string in the HTTP client.	18
6	Probe	No. of exceptions generated anomalously high while migrating mailboxes	Caused by a code-change to a mailbox migration components.	Keyword `migrat` matched a changed function's name.	12
7	Probe	Number of exceptions in the log file for a component **C** became abnormally high.	Support for a new data type was added in a component that made an API call to component **C**, but **C** does not support that data type.	The exception text for **C** contained the data type.	70
8	Probe	CPU Usage on a set of machines was anomalously high.	Reads and writes to a dictionary were not thread-safe. Multiple threads were reading from a dictionary while it was being modified, causing a CPU blowout.	No keyword matched.	89
9	Probe	POP and IMAP services started failing.	Dependencies were broken when a code commit changed a library that the POP and IMAP services used.	Keywords `Pop` and `Imap` matched in code changes.	110
10	Customer complaint	A client signing in via OAuth does not display calendar.	Client-side implementation was incompatible with the server-side commit.	Keyword `OAuth` matched the symptom and the server-side change.	39

Table 2: Examples of post-deployment bugs.

But we do recognize that anomaly detection algorithms are imperfect. Orca's build provenance graph helps find bugs even when the anomaly detection algorithm detects a bug well after a commit introduces the bug.

Orca does not handle bugs where the query does not have any context-specific information. Notable examples are performance issues, where the symptoms are, simply, out-of-memory exceptions or CPU-above-threshold exceptions. There exist other techniques [10, 48] in the literature which can debug such issues and therefore, can be used in combination with Orca. We discuss this further in Section 7.

4 Design

In this Section, we describe Orca's differential bug localization constructs in more detail. The input query to Orca is a symptom of the bug. This could be the name of the anomalous probe, an exception message, a stack-trace, or the words of a customer-complaint. The idea is to search for this query through *changes* in code or configurations that could have caused this bug. Thus the "documents" that the tool searches are properties that changed with a commit, such as names of files that were added, removed or modified, commit and review comments, modified code and modified configuration parameters. Orca's output is a ranked list of commits, with the most likely buggy commit displayed first.

First, we describe two novel constructs that we use for search-space pruning and search-space expansion respectively: differential code analysis, and the build provenance graph. Next, we describe the machine-learning based model of commit risk prediction which Orca uses to determine a rank-order of probably root-cause commits. Finally we provide a detailed description of the search algorithm.

4.1 Differential Code Analysis

Orca can search the entire code file to perform bug localization. But this approach can find false matches and drop Orca's precision, especially since we have found that every commit changes, on average, only about 20 lines of source-code per-file (including file additions), whereas the entire file can consist of hundreds of lines of code.

Another simple approach, on the other end of the spectrum, would be to search only the file names for matches, and not look into the code at all. While this is partially effective, our evaluation in Section 6 shows that this does not give us satisfactory recall.

We therefore employ a middle-path – differential code analysis – within Orca. Prior work has studied differential code analysis, albeit mostly at the semantic level [23]. It identifies relevant pieces of the code change that can potentially cause different behavior in the new version relative to the old version, but with reference to a specified *property*, such as null dereference, memory consistency, etc. Such techniques rely on compuationally expensive techniques such as differential symbolic execution [34] and regression verification [19].

We do not go with the semantic approach because (a) it is difficult to determine the full set of properties to capture all bugs in large-scale services, and (b) we would like to avoid the performance overhead of traditional static analyzers for differential analysis.

On the other hand, we could go with a complete *lexical* analysis on the differences, i.e., we can match terms in the query with terms in the difference, without any syntactic or semantic understanding. This approach will be relatively lightweight. However, it will miss several root-causes because very often, terms that match, such as protocol names, are parts of higher-level structures such as the method names and classes that have been modified. Consider Bug 6 in Table 2, for instance. the term `migrat` appears in the name of the function that has been changed, but not in text that has changed. Our techniques therefore need to identify *syntactic* constructs, such as methods and classes, that have changed.

Therefore, rather than going with a semantic or lexical analysis, we perform a syntactic analysis. We use the abstract syntax trees (AST) of the old and new version of the program to discover relevant parts of the source that have been *added*, *removed*, and *modified*.

Our analysis finds differences in entities of the following types: class, method, reference, condition, and loop. We create a "difference set" D of two ASTs, A_{old} and A_{new}, in the following way. Say e_i is the old version of an entity, and e_j the new version. Say t is the type of the entity. Then,

$$D = \begin{cases} d_{added} = \forall e \in A_{new} \mid e \notin A_{old} \\ d_{removed} = \forall e \in A_{old} \mid e \notin A_{new} \\ d_{changed} = \forall e_{new} \in A_{new} \mid type(e_{new}) = t \\ \qquad\qquad \wedge e_{new} \in A_{old} \wedge e_{new} \neq e_{old} \\ \qquad\qquad \wedge type(e_{old}) = t \\ d_{diff} = e_{old} \Delta e_{new} \mid e_{new} \in A_{new} \\ \qquad\qquad \wedge type(e_{new}) = t \wedge e_{old} \in A_{old} \\ \qquad\qquad \wedge e_{new} \neq e_{old} \end{cases}$$

Thus, the difference set D captures entities that have been *added*, *removed*, or *changed*. For all entities in D that have been changed, we also capture the differences (d_{diff}) between the two versions of the entity using a heuristic. For instance, say our heuristic detects that two lines of a function F have been changed. In addition to D containing the name of the function F, D also includes d_{diff}, which contains the entire text of the two changed lines: both the old version, and the new version.

We would like to point out that our syntactic approach to differential code analysis is not sound: we may detect changes even though there are none. For instance, consider the case where a function name changes completely, but the body remains unchanged. Our algorithm will treat this as a completely new function. While this could cause a precision drop in Orca since we are including more textual differences than actually exist, the algorithm does ensure that all changes are captured.

4.2 Build Provenance Graph

In Section 3.2, we have shown that a buggy commit to an older build may show symptoms only in a subsequent build. This could be because an inaccurate anomaly detection algorithm detects an anomaly too late. It may also be that a subtle bug manifests only in certain environments or only when a large number of users hit the service.

To accommodate this scenario, we expand our search to include previous builds that the symptomatic build is "derived" from. We maintain a *build provenance graph (BPG)* that captures dependencies between various builds along the axes of time and ring ID. Figure 2 shows an example fragment of a build provenance graph.

4.2.1 Construction

We now describe how we construct the build provenance graph. The BPG captures how builds are created and promoted in the different rings. Every build is represented as $B^i_{r,v}$, where i is a the build identifier, r is the ring identifier, and v is the version of the build within a ring.

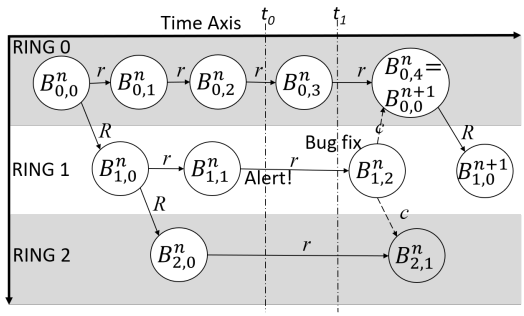


Figure 2: Example fragment of a build provenance graph.

Say a build spends a certain amount of time in Ring 0. If it is stable and shows no unhealthy behavior for a while, it is considered for promotion to the next ring. Build $B^n_{0,0}$ is one such build. It forks off build $B^n_{1,0}$ which runs in the next ring, Ring 1. Since $B^n_{1,0}$ is directly derived from $B^n_{0,0}$, any bugs that emerge in $B^n_{1,0}$ could potentially be due to commits originally made to $B^n_{0,0}$. Thus, we introduce an *inter-ring* edge between the two builds. Similarly, once a build is considered stable in Ring 1, it is forked off to Ring 2. This introduces the inter-ring edge between $B^n_{1,0}$ and $B^n_{2,0}$. For a given build identifier n, only one inter-ring edge can exist between two consecutive rings.

Meanwhile, developers make fresh commits $within$ each ring too, thereby creating the next build versions within the same ring. So in Figure 2, $B^n_{0,1}$ is derived from $B^n_{0,0}$, $B^n_{1,1}$ from $B^n_{1,0}$, and so on. We call the edges between these builds *intra-ring* edges. Several such intra-ring edges may exist in all rings though most are in Ring 0 which is the most dynamic and experimental ring.

At any given time, a vertical line drawn through the graph yields the *active build* within each ring at that time. For instance, at time t_1, the active builds are $B^n_{0,3}$, $B^n_{1,1}$, and $B^n_{2,0}$. For ease of explanation, we assume that at any given time there is only one active build in every ring though in reality there could be many.

We now explain a third edge-type called the *back-port* edge, shown in Figure 2 by dotted arrows. A critical bug that goes undetected may, with time, propagate to a large number of builds across all rings. Say at time t_0, such a bug causes an alert in build $B^n_{1,1}$. Say at time t_1, using the process described in Section 4.2.2, we localize the bug to a commit made to the earlier build $B^n_{0,0}$. The bug is fixed through a new commit c to the code of $B^n_{1,1}$, and this generates $B^n_{1,2}$, with an intra-ring edge between them.

We can see that since the bug originated in $B^n_{0,0}$, it also exists in the active builds within Ring 0 and Ring 2 that have been derived from it, i.e. $B^n_{0,3}$ and $B^n_{2,0}$. Consequently we apply the commit c, or we *back-port* it, to $B^n_{0,3}$ and $B^n_{2,0}$. This creates new builds $B^n_{0,4}$ in Ring 0 and $B^n_{2,1}$ in Ring 2. Thus we add two back-port edges with the label c from $B^n_{1,2}$ to $B^n_{0,4}$, and $B^n_{1,2}$ to $B^n_{2,1}$.

Finally, we describe how the build identifier n gets incremented in the build provenance graph. All builds across all rings that have the build identifier n are derived originally from $B^n_{0,0}$. Thus $B^n_{0,0}$ is called an *origin* build. With time, several new commits are applied in Ring 0. To ensure that these commits are fully deployed across all rings, a subsequent build from Ring 0 is chosen to be the next origin build. In the figure, this is $B^n_{0,4}$, which we rename as $B^{n+1}_{0,0}$, or the *next* origin build. All subsequent builds are now derived from this new origin build.

It can be seen that, barring backport edges, every node in the build provenance graph has only one incoming edge. This can be either an inter-ring or an intra-ring edge.

4.2.2 Traversal

We now describe how Orca uses the build provenance graph to expand its search-space and find potential buggy commits in older builds. Given a symptomatic build $B^i_{p,q}$, the purpose of the traversal is to find a list of candidate commits for the search.

We observe that Ring 0 is the most experimental of all rings. Builds in Ring 0 see a large number of significant commits. Consequently, our intuition is that, to localize a bug that appears in $B^i_{p,q}$, we should search all builds back to the the origin build $B^i_{0,0}$, which is in Ring 0. Thus using inter-ring and intra-ring edges, we backtrack from $B^i_{p,q}$ to the origin build $B^i_{0,0}$. In addition, we also include all back-ported commits to every build on the same path. Since every build has only one incoming inter-ring or intra-ring edge, there is only one such path from $B^i_{p,q}$ to $B^i_{0,0}$. The candidate list of commits to search will include all commits made to the builds on this path, and the back-ported commits on the same path.

We now explain this through an example with Figure 2. Say an alert is raised in $B^n_{2,1}$. Backtracking from $B^n_{2,1}$ to $B^n_{0,0}$ yields the set of commits $\{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}\})$, where $C(B_{i,j})$ is the set of commits that were made to build $B_{i,j}$. To this, we add c, which is a backported commit from $B^n_{1,2}$ to $B^n_{2,1}$, thereby giving us the final list of commits to search in. That is,

$$\Gamma = \{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}), c\}$$

4.3 Algorithm

In this Section, we describe Orca's search algorithm which uses differential code analysis and the build provenance graph. The Orca search algorithm consists of four

steps: 1) Query pre-processing where we perform tokenization, stemming and stop-word removal, 2) build graph traversal described in Section 4.2.2, 3) token-matching in code changes using Differential Code Analysis and 4) ranking and visualization of results. Our system runs differential code analysis and constructs the build provenance graph in the background periodically so that these tasks do not slow down the query response time.

Step 1: The search queries are symptoms of the problem, consisting of probe names, exception texts, log messages, etc. We first tokenize the terms in these symptoms by using a custom-built code tokenizer. This tokenizer uses heuristics that we have built specifically for code and log messages, such as splitting large complex strings along Camel-cased or Pascal-cased fragments. We also create n-gram based tokens since we have found that bigrams, such as `ImapTransfer` and `mailboxSync`, capture important information.

Next, we filter out irrelevant words also called stop-words [27] from these symptoms. Previous work has shown that logs have a lot of inherent structure [40]. For instance, all exception names have the suffix `Exception` and almost all log messages have a timestamp. Unlike conventional search-engines, even before we built Orca, we had access to about 8 million alerts consisting of probe names, exceptions and log messages from Orion's log store. We therefore perform stop-word removal on these to weed out commonly used or irrelevant terms such as `Exception` or timestamps. This step gives us a list of relevant "tokens" in the symptom. For each token t, we also maintain an *Inverse Query Frequency (IQF)* value [41] that we call t_{IQF}, obtained by analyzing Orion 's logs. t_{IQF} is calculated as (No. of queries/No. of queries in which token t appears). A high value of t_{IQF} implies that the token t is more important.

Step 2: We traverse the build provenance graph to find all builds related to the symptomatic build. From each build we discover, we enumerate all the commits that created the build. This leads us to the next step, which is matching tokens to files for each commit.

Step 3: Within a given commit C, for each file f and token t in the symptom, i.e for each tuple $T =< f, t >$, we search for the token in the difference set of the file, D_f. We use *TF-IQF* [41] as a "relevance" score, R_T^C, for each tuple. R_T^C is calculated as $n * t_{IQF}$ where n is the number of times the token t appears in difference set. This relevance score captures that the tuple $< f, t >$ is more relevant if the token t is very infrequent (i.e. t_{IQF} is very high), or if it appears many times in f.

We repeat this step for every token and file in the commit. At this point, we have file-level relevance values. Note though that we perform commit-level bug localization. Thus, we now aggregate the relevance values across all files and tokens to get one relevance value for the commit C, that we call R^C. So,

$$R^C = \underset{T}{\Theta} R_T^C \tag{1}$$

where Θ is an aggregation function such as such as `Max`, `Avg` or `Sum`. We show in Section 6 that the MAX function provides the best results in our deployment.

Step 4: Finally, Orca returns a list of commits in decreasing-order of their relevance. However, in deployment, we found that ranking solely based on decreasing order of relevance was not enough. Quite often, more than one commit had the same relevance score because several commits made at similar times matched the search terms equally.

We therefore build a machine-learning model that predicts commit risk to break the tie between commits that have the same relevance. This model uses ideas from a vast body of prior-work in this space in the Software Engineering community [5,20,30,32]. However, we believe ours is the first tool to apply such a risk prediction model to bug localization at the commit-level.

We have built a regression tree-based model that, given a commit, outputs a risk value for it which falls between 0 and 1. This is based on data we have collected for around 93,000 commits made over 2 years. Commits that caused bugs in deployment are labeled "risky" while those that did not, we labeled "safe". We have put in considerable effort into engineering the features for this task. The features that we input to the learner roughly fall into four categories:

- **Developer Experience**. Developers who are new to the organization and to the code-base tend to create more post-deployment bugs. Hence, we associate several experience-related features with each commit.
- **Code ownership.** We found that certain files that were mostly changed by a single developer caused fewer bugs than files that were constantly touched by several developers. Hence, for each commit, we use features to capture whether it touched files with very few owners or many owners.
- **Code hotspots.** Certain code-paths, when touched, tend to cause more post-deployment bugs than others. Some of our features capture this fact.
- **Commit complexity.** Several features, such as file types changed, number of lines changed, and number of reviewer comments capture the complexity of the commit.

Thus, for commits that have the same relevance score based on the terms match, we use the commit risk as a

secondary sort key to obtain a rank-order for Orca's output.

4.3.1 Example

We shall use the following example to illustrate how the search algorithm works. Say commits C_1 and C_2 create a build B. Say a probe called `LdapAuthProbe`, that monitors the LDAP authentication service, starts throwing exceptions of type `AuthFailedException`. Let us also say that the bug was caused by commit C_2 that erroneously modified a function `LdapRequestHandler` in a class `LdapService`, declared and defined in a file `LdapService.cs`. Say C_1 modified an `Imap` protocol implementation in a file `Imap.cs`.

The query to Orca is "`LdapAuthProbe AuthFailedException`". First, we tokenize and stem the query, and remove stop-words `Failed` and `Exception`. This yields the tokens `Ldap`, `Auth` and `Probe`. The word `Probe` occurs very frequently across all symptoms and therefore receives a very low IQF score, whereas `Ldap`, being a specific protocol, gets the highest IDF score. `Auth`, being somewhat more frequent than `Ldap`, receives a slightly lower IQF score.

We leave out build graph traversal for the sake of simplicity. Therefore, our list of candidate commits are only C_1 and C_2. In our example, the token `Ldap` will match both the class name, `LdapService`, and the function modified, `LdapRequestHandler`. Therefore the value of relevance for this is $2*$ the IDF value of token `Ldap`. C_2 will also find a match with token `Auth`. C_1, however, does not match any of these tokens. Our ranking algorithm will therefore choose C_2 over C_1 and, as the highest-ranked result, it will show the filename `LdapService.cs` and token `Ldap` rather than `Auth`.

5 Implementation

We have implemented Orca in a combination of C# (using .NET Framework v4.5) and SQL. We use the FastTree [16] algorithm within the ML.Net [31] suite for our commit risk prediction. Currently, the implementation is approximately 50,000 lines of code. In this Section, we briefly describe our implementation of Orca, and the various user interfaces we expose for OCEs.

5.1 Data Loaders

Since Orca requires information about various different parts of the system – source-code, builds, deployment information and alerts – a significant part of our implementation are data loaders for these different types of data. Figure 3 shows an architectural overview of the implementation. At the heart of Orca is a standard SQL database. This database is populated by data loaders at a predefined frequency. We now describe the different data loaders we use.

Source Data We implemented loaders for various source-control systems such as GIT and others internal to our organization. These loaders ingest source-code, code-versions and histories. The differential code analysis algorithm uses data from this loader.

Builds Data about builds resides in multiple big-data logs. We have build loaders that interface with several big-data logging systems to load build-specific information into our SQL database. We use this loader to construct the build provenance graph.

Deployment and Machine Data We load logs created by continuously monitoring the state of all machines within all rings. The state includes information about the current status of the machine (healthy/unhealthy), along with the information on the build version running on the machine.

Alerts are loaded from existing databases. Today, Orca supports multiple data sources for loading alert information.

5.2 Background Analyses

As our data loaders periodically load new data into the SQL database, we periodically initiate differential code analysis, build provenance graph construction, IQF calculation, and the commit risk prediction used in Section 4.3. If needed, the frequency of an analysis can be changed to make it more/less frequent. For example, the IQF calculation runs once a week, while commit risk prediction runs once every day. The other two processes run once every hour. Finally, it should be noted that all analyses that have been developed and deployed within this system are agnostic of the data source. The SQL database schema is normalized; thus, providing the same interface to all analyzers irrespective of the data source.

5.3 API Implementation

We now describe the Orca API and its implementation. Each Orca request is processed in real-time by the core Orca engine, and results are returned in JSON format. Clients decide on the relevant parts of the return result and how to display them. We make use of a Redis Cache [8] for improving our lookup times associated with data that is static. This includes data about the

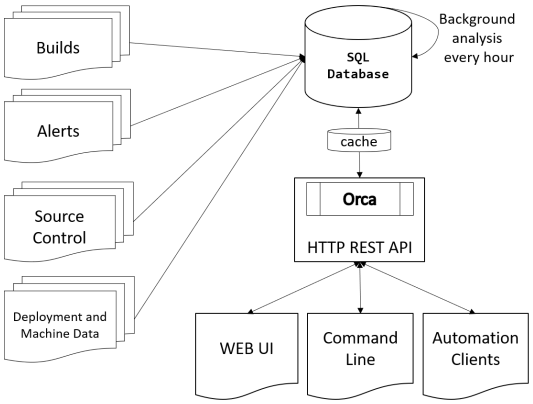


Figure 3: Implementation of Orca.

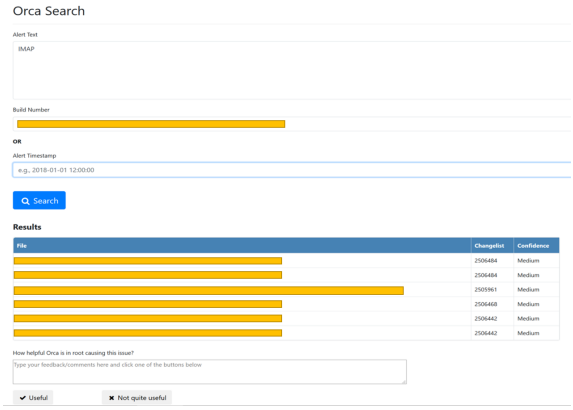


Figure 4: Web based UI of ORCA for Orion.

builds, files, source-code, difference sets, and the build provenance graph. Internally, the Orca system records all requests made and any feedback provided, which we use for learning and improvement.

All our services, servers, databases, and caches are implemented and operated using Azure as both a Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) provider.

5.4 Usage

Orca can be used in two ways: OCEs can use it to make *ad-hoc queries* interactively, or an alerting infrastructure can query Orca to get a list of suspect commits for an alert and include it in the alert itself.

For the OCEs, we have built a web-based UI that allows them to enter the details of their query and view the results. Figure 4 shows a screenshot of the UI with sensitive information removed. We have also built a PowerShell® *cmdlet* with which the OCE can interact with through a command-line interface.

To integrate Orca with alerting infrastructure, we have built an API that the alerting system can query directly. Currently, this mechanism is being used by multiple groups that are generating alerts within the Orion group. The web-based UI and *cmdlet* tend to be used by OCEs when new information such as log text or exception text has been discovered after the original alerts were generated.

Today, Orca has been deployed on multiple code-bases for six large-scale services within our enterprise. The combination of data loaders used in each code-base is slightly different and unique, but fully operational and functional.

6 Evaluation

In this section, we provide results we obtain by evaluating Orca. First, we evaluate Orca's result quality, i.e. how often it attributes a bug correctly to the right commit. Next, we evaluate how much effort an OCE saves by using Orca. Third, we evaluate the performance of Orca and the savings we get by using a Redis-based cache.

Since its deployment with Orion in October 2017, the Orca API has been invoked 4400 times to debug issues within the Orion service. Unfortunately, there is no central location where OCEs retrospectively log information about buggy commits. To evaluate how well Orca localizes a bug, we not only need the complete symptom of the bug, such as error messages or exceptions, but also the root-cause commit. Consequently, we begun a manual process towards gaining this information for as many bugs as we could. We interviewed multiple OCEs and manually analyzed source-code, bug-reports and email-threads.

By performing this exercise, we collected complete information for 48 of these bugs. These bugs vary greatly in characteristics. While some were inadvertently introduced by a single line-change, others were caused by complex dependencies between components.

6.1 Result Quality

To measure the quality of Orca's results, we interviewed several OCEs about how they would quantify result quality. Based on these interviews we determined that it is important that we find the buggy commit in as many cases as possible. This is captured by the *Recall*, i.e. the fraction of bugs where we found the buggy commit in any position in our results. The OCEs also told us that they linearly scan the list of commits that Orca provides, hence the closer the correct commit is to the top, the more time they save. To capture this, we use the *Mean Recip-*

	No. of results = 5			No. of results = 10			No. of results = 20		
Agg.fn.	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL
MAX	0.65(31)	0.60(29)	0.63(30)	**0.69(33)**	**0.77(37)**	**0.77(37)**	0.69(33)	0.77(37)	0.77(37)
SUM	0.52(25)	0.52(25)	0.60(29)	0.71(34)	0.67(32)	0.77(37)	0.73(35)	0.79(38)	0.77(37)
AVG	0.60(29)	0.46(22)	0.58(28)	0.67(32)	0.73(35)	0.77(37)	0.69(33)	0.75(36)	0.77(37)

Table 3: We use recall to evaluate applying differential code analysis alone (DCA), differential code analysis with build provenance graph (DCA+BPG), and differential code analysis with build provenance graph and commit risk (ALL). Numbers in parentheses are the number of bugs correctly localized. We evaluate aggregating by MAX, SUM, and AVG.

	No. of results = 5			No. of results = 10			No. of results = 20		
Agg.fn.	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL
MAX	0.41	0.38	0.39	**0.44**	**0.38**	**0.42**	0.44	0.38	0.42
SUM	0.42	0.36	0.40	0.44	0.38	0.43	0.44	0.39	0.43
AVG	0.36	0.25	0.38	0.37	0.29	0.41	0.37	0.29	0.41

Table 4: We use MRR to evaluate applying differential code analysis alone (DCA), differential code analysis with build provenance graph (DCA+BPG), and differential code analysis with build provenance graph and commit risk (ALL). We also evaluate aggregating by MAX, SUM, and AVG.

rocal Rank (MRR) [28]. MRR is the most suitable metric since we assume there is only one buggy commit that causes the symptom. MRR is calculated as $\frac{1}{n}\Sigma_{i=1}^{n} 1/r_i$, where n is the number of queries, r_i is the rank of the buggy commit for query i. If Orca is unable to find the correct commit, we assume r_i is infinity, i.e. we add 0 to the sum total.

We first evaluate differential code analysis (DCA) only. Next, we add the build provenance graph, without the commit risk-based ranking (DCA+BPG). Finally, we add commit risk prediction and evaluate Orca using all the techniques described in the paper (ALL). Tables 3 and 4 show the results for various combinations of parameters and features. We have varied the number of results we return as part of the Orca API to evaluate the quality for 5, 10 and 20 results, and we have evaluated Orca for different aggregation functions (Θ in Equation 1): MAX, SUM and AVG.

To evaluate Orca, we ask three questions:

- **How much value does differential code analysis add?** We wanted to understand whether looking into code was necessary.
- **How much value does the build provenance graph and commit risk prediction add?** We wanted to understand in what number of cases the build graph helped improve result quality. In addition, we wanted to see whether our rank-order based on commit-risk helped improve our results.
- **How should we aggregate the file-level relevances to commit-level?** Equation 1 in Section 4.3 described how we need to aggregate file-level relevance value into one value at the commit-level.
- **How many results should we show in the Orca UI?** When asked, the OCEs mentioned that they would not want to see beyond 10 results for each query. Hence, we wanted to evaluate the trade-off between the number of results shown and the recall and MRR.

We now answer each of these questions in order.

The build provenance graph adds 8% to the recall. Observe the data in bold in Table 3. DCA alone localizes 33 bugs for a recall of 0.69. Adding the build provenance graph helps us localize 4 more bugs correctly thereby increasing our recall to 0.77. While at first glance, this may appear to be a small increase, OCEs find it significantly more difficult to attribute bugs that occur in older builds, and therefore the value of finding these 4 bugs eases the OCE's workload considerably. We show this quantitatively in Section 6.2.

Adding commit risk-based ranking improves MRR by 11%. Observing the data in bold in Table 4, one can see that adding the build provenance graph reduced our MRR from 0.44 to 0.38. This happens because the build provenance graph increases our search-space significantly, and this increases the number of false-positive matches between terms and commits. Here, adding our secondary rank order based on commit risk improved our results by restoring the MRR value to 0.42.

The MAX and SUM aggregation functions per-

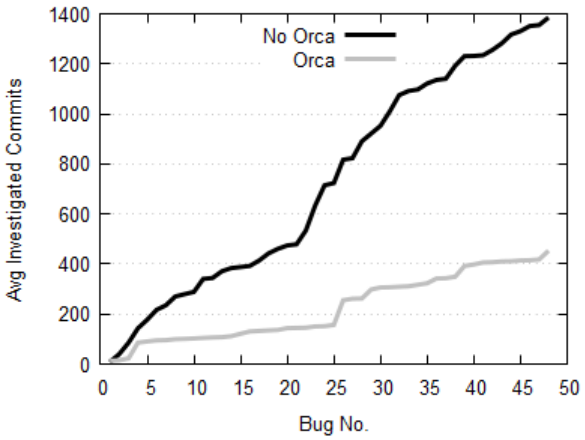


Figure 5: For all 48 bugs, a cumulative distribution function of the expected number of commits that the OCE investigates without Orca and with Orca.

form better than AVG. While the buggy code-changes match some very high-relevance tokens, several lower-relevance tokens match these code-changes too. Hence taking the average value across all matches dilutes the high-value token matches, therefore reducing both recall and MRR. Such a dilution does not happen if we use MAX or SUM. We choose MAX in our implementation.

Showing 10 results seems a good trade-off between result quality and UI succinctness. We evaluated results while setting the number of results shown as 5, 10 and 20. We find that with 10 results we achieve close to our best recall and MRR values.

With 10 results and using MAX, we obtained a recall of 0.77, i.e. we found the root-cause in 37 out of 48 cases. The MRR was 0.42. We also studied the matched terms for these 37 bugs and found that term-similarity serves as a good proxy to capture different types of bugs. Table 2 showed that matched terms fall roughly into four categories: they either match a component name, a function that the component performs such as `migrat` or `suggest`, data types, or protocol names such as `Imap`. We found in our case that of the 37 correctly localized cases, in 14 cases the token was a component name, in 17 cases it captured the function being performed, in 4 cases it matched a protocol name, and in the remaining 2 cases, the match was on a data type. Therefore term-similarity is quite versatile: it helps us catch a variety of bugs.

Of the 11 cases that we could not localize, in 1 case the issue was related to performance. In 4 cases, the problem was because a configuration setting. changed, and that triggered the use of code that was committed much earlier than our build provenance graph covered. In future work, we therefore plan to include all configuration settings in our differential analysis. In the remaining 6 cases, term similarity just did not capture the high-level semantics of the bug, and static or dynamic analysis may be required.

6.2 Reduction of OCE Workload

We now investigate the effect of Orca on reducing the OCE's workload. There are multiple ways to measure this. One way is to measure the amount of time saved by Orca for the OCE. Another is to determine the decrease in the number of commits that the OCE needs to manually investigate, both with and without Orca. We chose the latter metric because it allows us to quantify the reduction of OCE workload both at the level of every individual bug and as an aggregate. The former metric, i.e. the amount of time saved, can only provide us an aggregate across all alerts. Moreover, unless we shadow OCEs over an extended period of time, it is difficult to accurately quantify the time saved.

Given a bug, an OCE will investigate an average of $c/2$ commits to localize it, where c is the number of commits in the buggy build. If the OCE uses Orca, they need investigate at most r commits, where r is the rank of the correct commit in the results that the UI shows. If Orca does not find the correct commit, then apart from the 10 commits that Orca shows, the OCE needs to investigate an additional $(c-10)/2$ commits in expectation.

Figure 5 shows the number of commits investigated with and without Orca, for all 48 bugs that we evaluated, as a cumulative distribution function. Over all 48 bugs, without Orca, the OCE investigates a median of 22.75 commits, and an average of 28.9 commits. With Orca, she investigates a median of just 3.5 commits and an average of only 9.4 commits. Therefore, using Orca causes a $6.5\times$ reduction in median OCE workload and a $3\times$ (67%) reduction in the OCE's average workload. For the 37 bugs Orca localizes, this reduction factor in the average is much higher, i.e. $9.7\times$. For the 4 bugs that were caught only because of the build provenance graph, the OCE had to investigate an average of 59.4 commits without Orca, and only 1.25 commits with it. This is a $47.5\times$ improvement. These numbers point out the benefits that the build provenance graph gives us, and the benefits overall of using Orca for commit-level bug localization.

6.3 Performance

Finally, we evaluate the performance of Orca. We run Orca on a 32 core, 2GHz Intel Xeon E7-4820 CPU with 64 GB memory. We ran all 48 queries in sequence to obtain these results. First, we evaluated the effect of the Redis cache on Orca's average query response time. Using

Redis, the average response time reduced from 12.4 seconds to 5.97 seconds, a gain of 51.8%. Next, we varied the degree of parallelism in Orca from 32 to 128 using the C# MAXDOP parameter [37] and noticed a significant effect on average query response time. With parallelism set to 32, the average response time is 30.94 seconds, whereas with parallelism of 128, it is 5.97 seconds. Our evaluation shows that there is a significant potential for parallelizing Orca further, thereby catering to many more queries and providing lower response times. Therefore we can effectively scale Orca out to more services within our enterprise without loss in performance.

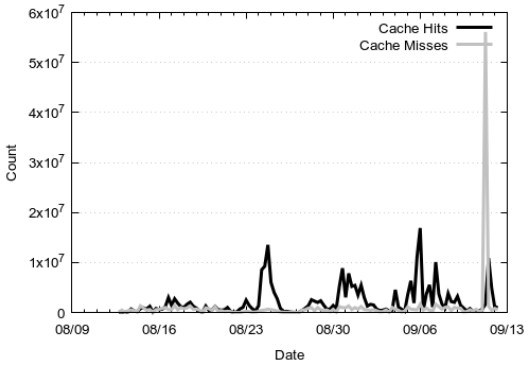


Figure 6: The efficacy of using Redis in Orca's deployment.

Figure 6 shows the effect of using Redis with Orca's deployment over two months, starting from August 2018. On average, the number of hits is 2.2 times the number of misses. This shows that users of the Orca API make queries that have locality and therefore benefit greatly from the use of the Redis cache. This is to be expected as successive queries to Orca will be highly likely for the same builds and therefore will access similar difference sets.

7 Discussion

In this section, we first discuss the generalizability of Orca to other services. We then discuss the limitations, and how we intend to address these in the future.

7.1 Generalizing Orca

Fundamentally, in a CI/CD pipeline, to save time and resouce-usage, services combine various commits before they build, test and deploy. Performing these procedures after every commit would be prohibitively expensive. Since such an aggregation of commits is absolutely necessary, so is the need for a tool that localizes bugs at the commit-level, such as Orca.

Though we describe Orca in the context of a large service and post-deployment bugs, we believe the techniques we have used also apply generically to many Continuous Integration/Continuous Deployment (CI/CD) pipeline. This is based on our experience with multiple services that Orca is operational on within our organization. Orca needs expressive symptoms as input. Modern-day services use rich monitoring systems enabled by infrastructure such as Nagios [33] and AlertSite [2] which can provide probe-level symptoms of problems.

7.2 Future Work

Two types of bugs that Orca currently cannot localize are performance and configuration issues. Addressing these bugs, therefore, are immediate next-steps.

To deal with performance-related bugs, we plan to incorporate better anomaly detection algorithms, and correlate anomalies with code-changes that could potentially cause them. We believe that the principal idea of Orca, i.e. keyword match, can also be applied to bugs that arise from faulty configuration settings. Therefore, we plan to include configuration-based difference sets into our search engine.

8 Conclusion

In this paper, we described Orca and the differential bug localization algorithm. Orca uses differential code analysis and the build provenance graph to find buggy commits in large-scale services. Orca is deployed with a large email and collaboration platform. We have shown that Orca finds the correct buggy commits in about 77% of bugs that we studied, and causes a $3\times$ reduction in the work done by the OCE. We have also shown that Orca is efficient, accurate and easy to deploy.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPlan Notices*, volume 25, pages 246–256. ACM, 1990.

[2] AlertSite - Application Performance Monitoring Tools. `https://smartbear.com/product/alertsite/overview/`. Accessed: 2018-09-25.

[3] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: A recommender system for debugging. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software*

Engineering, ESEC/FSE '09, pages 373–382, New York, NY, USA, 2009. ACM.

[4] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of FSE*, 2011.

[6] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[7] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 2–9. ACM, 2002.

[8] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[9] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 217–231, Berkeley, CA, USA, 2014. USENIX Association.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[12] H. Cleve and A. Zeller. Locating causes of program failures. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 342–351. IEEE, 2005.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[14] O. De Moor, M. Verbaere, and E. Hajiyev. Keynote address:. ql for source code analysis. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 3–16. IEEE, 2007.

[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, Västeras, Sweden, 2014.

[16] FastTree (Gradient Boosted Trees). `https://docs.microsoft.com/en-us/machine-learning-server/r-reference/microsoftml/rxfasttrees`. Accessed: 2018-08-23.

[17] Git - Version Control System. `https://git-scm.com/`. Accessed: 2018-09-25.

[18] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[19] B. Godlin and O. Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471. ACM, 2009.

[20] A. Hindle, D. M. German, and R. Holt. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of MSR*, 2008.

[21] I. Infonetics. The cost of server, application, and network downtime: Annual north american enterprise survey and calculator. *Computing*, 2016.

[22] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007.

[23] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[24] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.

[25] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports

(n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[26] B. Liblit, A. Aiken, M. Naik, and A. X. Zheng. Scalable statistical bug isolation. In *In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26. ACM Press, 2005.

[27] H. P. Luhn. Key word-in-context index for technical literature (kwic index). In *Proceedings of the 136th Meeting of the American Chemical Society, Division of Chemical Literature*, 1959.

[28] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[29] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[30] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *ICSE SEIP Track*, 2017.

[31] ML.NET Machine Learning Framework. `https://www.microsoft.com/net/learn/apps/machine-learning-and-ai/ml-dotnet`. Accessed: 2018-08-23.

[32] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of ESEM*, 2007.

[33] Nagios - The Industry Standard In IT Infrastructure Monitoring. `https://nagios.org`. Accessed: 2018-09-19.

[34] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[35] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[36] M. Rath, D. Lo, and P. Mder. Replication data for: Analyzing requirements and traceability information to improve bug localization, 2018.

[37] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition, 2012.

[38] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[39] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[41] J.-M. Yang, R. Cai, F. Jing, S. Wang, L. Zhang, and W.-Y. Ma. Search-based query suggestion. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 1439–1440, New York, NY, USA, 2008. ACM.

[42] K. C. Youm, J. Ahn, J. Kim, and E. Lee. Bug localization based on code change histories and bug reports. In *Software Engineering Conference (APSEC), 2015 Asia-Pacific*, pages 190–197. IEEE, 2015.

[43] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[44] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association.

[45] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[46] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk

failure diagnosis and pattern detection for azure. In *In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[47] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581, New York, NY, USA, 2017. ACM.

[48] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644, Berkeley, CA, USA, 2014. USENIX Association.

[49] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24. IEEE Press, 2012.

Differential Energy Profiling: Energy Optimization via Diffing Similar Apps

Abhilash Jindal and Y. Charlie Hu
Purdue University and Mobile Enerlytics, LLC

Abstract

Mobile app energy profilers provide a foundational energy diagnostic tool by identifying energy hotspots in the app source code. However, they only tackle the first challenge faced by developers, as, after presented with the energy hotspots, developers typically do not have any guidance on how to proceed with the remaining optimization process: (1) Is there a more energy-efficient implementation for the same app task? (2) How to come up with the more efficient implementation?

To help developers tackle these challenges, we developed a new energy profiling methodology called *differential energy profiling* that automatically uncovers more efficient implementations of common app tasks by leveraging existing implementations of similar apps which are bountiful in the app marketplace. To demonstrate its effectiveness, we implemented such a differential energy profiler, DIFFPROF, for Android apps and used it to profile 8 groups (from 6 popular app categories) of 5 similar apps each. Our extensive case studies show that DIFFPROF provides developers with actionable diagnosis beyond a traditional energy profiler: it identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks, and the call trees of tasks it extracts further allow developers to quickly understand the reasons and develop fixes for the energy difference with minor manual debugging efforts.

1 Introduction

Despite the prevalence of smartphones, the user experience has remained severely limited by their battery life. As such, major mobile platform vendors such as Apple and Google have taken initiatives encouraging app developers to take effort optimizing their apps [7, 26].

The typical development cycle for optimizing the energy drain of mobile apps is similar to that for optimizing the running time of traditional software – iterating the process of (1) finding hotspots in the app source code that contribute to a significant portion of the total app energy drain, and then (2) determining whether and how the energy hotspots can be restructured to drain less energy.

However, modern mobile apps are highly complex, easily consisting of millions of lines of source code and third-party software, and interacting with the OS-provided frameworks in complex ways. Without the help of automatic tools, even finding energy hotspots in the app source code by developers would be very hard.

To this end, mobile app energy profilers (*e.g.,* [32, 31]) made a major step forward by providing a foundational energy diagnostic tool that automatically identifies energy hotspots in the app source code. However, these profilers only help with the first step of the app energy optimization process, because after presented with the energy hotspots, developers typically do not have any guidance on *whether* and *how* the energy hotspots can be restructured to drain less energy.

To help app developers with this remaining challenge in the energy optimization process, in this paper, we develop a new energy profiling methodology called *differential energy profiling* (or *energy diffing* for short) that can automatically uncover more efficient implementations of common app tasks, and in doing so, not only determines whether an energy-hotspot code segment can be optimized, but also gives hints on how to optimize it.

The basic idea behind differential energy profiling is intuitive: if we can find a set of similar apps by different developers that implement many identical app tasks, chances are the implementations differ and will have different energy footprint. Directly comparing their source-code energy profile generated by an energy profiler should expose more efficient implementation from the less one for the same app tasks.

In this work, we first make three key observations about the uniqueness of the mobile app marketplace and common mobile app development practice: (1) Because of the low barriers to entry of app development, for every popular app in the app market, there are typically a few dozen competing apps that implement similar or identical app functions or app features. (2) Using a traditional energy profiler, we profiled 8 selected app groups from 6 popular app categories from Google Play, each consisting of 5 similar apps and 5 different versions of one of them, and we found similar apps can differ significantly in energy drain in performing similar app functions. (3) We further observe that mobile apps make heavy use of the common framework services provided by modern

mobile OSes such as the Android framework, and our profiling analysis of the above 8 app groups has shown similar apps in each group share 38.6% to 81.9% of the same framework method calls and spent 44.0% to 96.7% of their app energy drain in calling framework services.

Observations (1) and (2) suggest it is possible to learn more efficient implementation of the same app task by comparing the energy profiles of similar apps, but if apps have very different source code structures, such comparison may not be effective. Observation (3) affirms such comparison of similar apps is actually meaningful and potentially effective.

We then present the design and implementation of such a differential energy profiler, DIFFPROF. Developing DIFFPROF faces three challenges: (1) What should be the diffing granularity? (2) How to identify the diffing units in the source-code energy profiler output of each app? (3) How to actually diff the energy profiles of similar apps? We address these challenges as follows:

(1) Using app tasks as the diffing granularity. We argue following the widely adopted modular programming principle, an app is typically structured to implement a number of app features or tasks. Since the ultimate goal of energy diffing is to uncover more efficient implementations of app tasks, the ideal diffing granularity that most directly helps the developers should be an app task.

(2) Characterizing how app tasks manifest in call trees. Diffing at the app task granularity requires identifying app tasks in the call tree output by a source-code energy profiler. To address this challenge, we examine the call trees for top 100 non-game apps and find that app tasks manifest themselves as Erlenmeyer flask-shaped slices (denoted as EFLASKS) represented in (call path, framework-method, subtree) tuples where the call path identifies the context of the task, the framework-method is used to invoke the framework service to accomplish the task, and the subtree captures the particular execution of the framework service.

(3) An efficient EFLASK matching algorithm. We give insights on how and why different implementations (EFLASKS) of the same app task differ which motivates the need for approximate EFLASK matching. We develop to our knowledge the first EFLASK-shaped tree slice matching algorithm that accurately finds similar EFLASKS corresponding to the same app task.

To demonstrate its effectiveness, we implemented DIFFPROF on top of a state-of-the-art energy profiler EPROF [32] for Android, and compared it to EPROF in profiling 8 groups (from 6 popular app categories in Google Play) of 5 similar apps each. We show DIFFPROF accurately identifies matched tasks that account for 79% of the app total energy drain on average as well as unique tasks (21% of total energy on average), in similar apps.

Further, we conducted 12 case studies to show that DIFFPROF provides developers with actionable diagnosis beyond a traditional energy profiler: (1) When EPROF identifies energy bottlenecks, they may be necessary or not inefficient; DIFFPROF identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks; (2) The EFLASK of tasks extracted by DIFFPROF further shows the details of the more efficient implementation, which allows the developer to quickly understand the reasons for the energy difference with minor manual debugging efforts (*e.g.,* setting breakpoints) since the developer did not author the similar app. Out of the 12 inefficient or buggy implementations in 9 apps, 3 of which have already been confirmed by developers, and removing them reduces app energy drain by 5.2%–27.4%.

This work makes the following contributions:

- It presents differential energy profiling, which tackles a key challenge faced by app developers in optimizing app energy drain - determining whether and how energy hotspots in app source code can be optimized, by identifying and comparing different implementations of the same tasks in similar apps.
- It presents DIFFPROF, an energy diffing tool for Android mobile apps. It describes DIFFPROF's implementation and the core algorithm that finds approximate matching of Erlenmeyer flask-shaped slices in calling context trees of similar apps, and demonstrates its benefits over traditional energy profilers.

2 Key Insights

The DIFFPROF design is motivated by three key insights we make about the mobile app market.

2.1 Competing/similar apps are abundant

Our first observation is about a unique phenomenon of the mobile app marketplace: *(O1) for every popular app, there are typically a few dozen competing apps that implement similar or identical app functions or app features.* The top 100 non-game apps in Google Play belong to 34 functionally similar app groups and each of these categories consists of many competing popular apps. Table 1 lists 8 such similar app groups with apps in the top 100 as well as outside the top 100 apps; the majority of them have 50M+ downloads.[1] We see that many groups include over a dozen similar apps each. Moreover, similar apps, *e.g.,* competing apps such as Pandora and Spotify, or a popular app (Candy Crush Saga) and its dozens of clones, typically have similar user interactions. For example, the music playback screens of all music streaming apps have an album cover image, the song and the album title, a progress bar, elapsed and remaining time text, and buttons to control music playback, and every app performs music playback.

Table 1: Eight groups of similar apps from top 100 non-game apps, their competitors, and energy drain measurement. "*": Popular but not a top 100 app, "+": Pre-installed app.

App Category	App Group	Similar/Competing Apps	Max/min energy ratio	Perc. energy in framework
Communication	Messaging & calling	Whatsapp, Google Hangouts+, Facebook Messenger, BBM, Line, Wechat, Viber, Skype, Tango, Whatscall, Telegram, TextNow, imo	8.0	60.2% - 90.3%
	Email	Yahoo Mail, Gmail, Outlook, Android mail+, Aqua Mail*, Email For Any*, MailRU*, myMail*	4.6	56.6% - 90.9%
Music & Audio	Music streaming	Spotify, Pandora, Soundcloud, iHeartRadio, Youtube Music, Free music, Napster, Google Play Music+, Apple Music*	4.2	49.6% - 93.8%
Personalization	Launcher	GO Launcher, CM Launcher 3D*, APUS Launcher*, Solo Launcher*, Hola Launcher*	3.1	44.0% - 93.8%
Productivity	File explorer	ES*, FX*, Solid*, File explorer*, File manager*	5.3	89.3% - 94.9%
Shopping	Shopping	Wish, eBay, Amazon, Walmart, AliExpress, Kohl*, letgo*	3.2	83.1% - 96.7%
Tools	Antivirus	Supo Security, CM Security AppLock AntiVirus, 360 Security, AVG AntiVirus, DU antivirus, Mobile Security & Antivirus*, Kaspersky Antivirus Security*	2.8	53.5% - 91.3%
	Cleaning	Clean Master, DFNDR, Fast Cleaner - Speed Booster, Turbo cleaner, Power clean Lionmobi, OK clean lite, DU speed booster & cleaner*, Ccleaner*	3.6	78.5% - 93.9%

2.2 Similar apps differ in energy drain

Given the abundance of similar apps for every popular app, we next ask the question: how do they stack against each other in energy drain, in performing similar app functions? To answer this question, we profiled the similar apps in the 8 popular app categories on a Nexus 6 phone running Android 6.0.1 while connected to WiFi.

We use automated tests to perform identical actions on the similar apps in each group and measure the energy drained by these actions using EPROF. In particular, we use `UI Automator`, the Android black-box UI testing framework, which does not require app source code.

For each group of similar apps, we first write a generic base test that interacts with common UI elements. Next, for each app in the cluster, we launch the app on the phone and find the unique ids of all the UI elements involved in the base test using Android's `uiautomatorview` tool. Finally, we run the base test with app-specific UI element ids, thus performing homogeneous interactions across similar apps. The specific tests for the 8 app groups are listed in the sub-captions of Figure 1.

Figure 1 contrasts the total energy drain of 5 similar apps and 5 versions of 1 app under the same user interactions in each of the selected 8 app groups from Table 1. We observe that the maximal to minimal energy drain across the 5 apps in each group range between 2.8x to 8.0x, as shown in Table 1. We thus draw our second observation that *(O2) similar apps easily differ significantly in energy drain in performing similar app functions.*

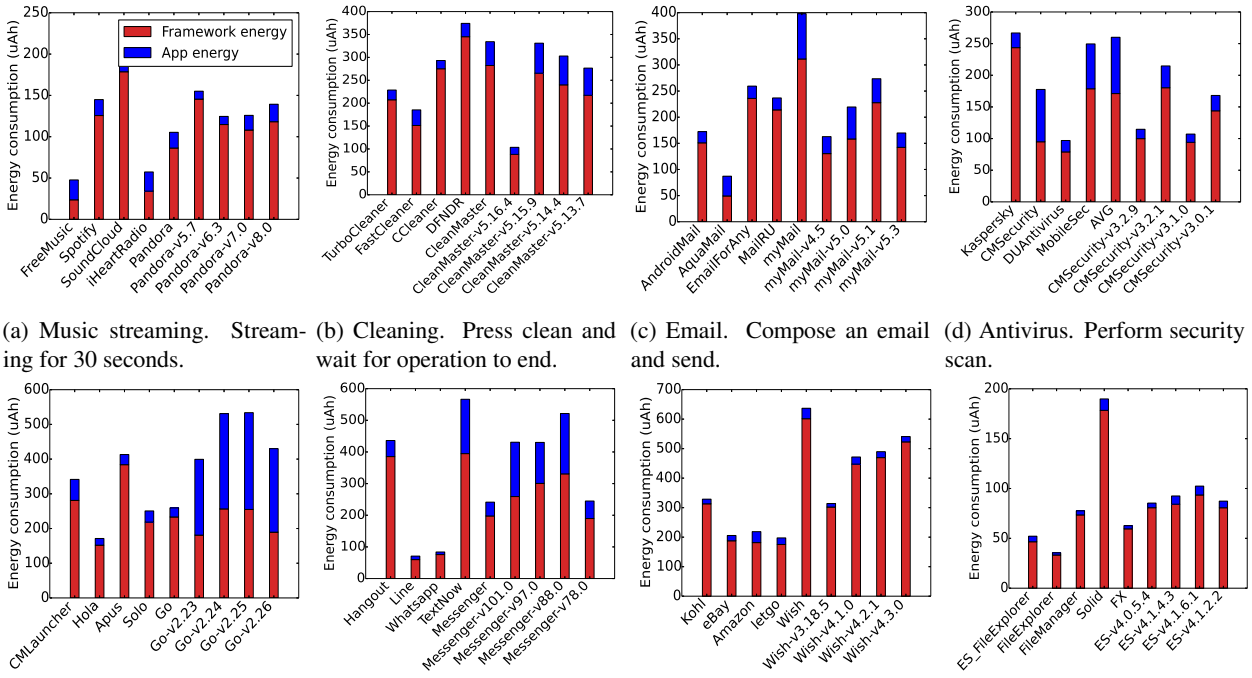
The above observation suggests that directly comparing the energy footprint of similar apps at the source-code level is promising to diagnose energy hotspots. However, such comparison will be fruitful only if their source code have significant overlap.

2.3 Framework services dominate app energy drain

Our next observation is that mobile apps make heavy use of the common framework services provided by modern mobile OSes such as the Android framework. To simplify app programming, such frameworks implement and export to apps many services that implement commonly performed tasks, *e.g.,* the Android framework provides `LocationManager`, `DownloadManager`, `MediaPlayer`, and `WindowManager`, among others. Typically, an app presents requirements via configuration parameters to the services, and the services then perform the low-level work on the app's behalf. We hypothesize that the heavy usage of framework services leads to a high percentage of app energy drain occurring in these common services and the framework methods called in similar apps have a high overlap.

To confirm this hypothesis, we use EPROF to decouple the energy spent in app methods from those spent in framework services. First, we run `dexinfo` [5] on all the framework jar files located in `/system/framework/` on the phone to identify all the framework packages such as `android.view`, `dalvik.system` and `java.math`. Next, for each app, we identify all the framework methods in its energy profiling output belonging to these framework packages. Finally, we aggregate their energy drain to compute the total framework energy drain. The remaining energy drain is marked as app energy drain.

Our results (details for only 4 app groups are shown in Figure 2 due to page limit) show that the apps in the 8 app groups have significant pairwise overlap in the framework methods called during the profiling run, between 38.6% and 81.9% (61.7% on average). Further, Table 1 shows that a significant portion of the total energy of the apps in each group was spent in framework API calls, ranging between 44.0% to 93.8% for Launcher apps to



(a) Music streaming. Streaming for 30 seconds.

(b) Cleaning. Press clean and wait for operation to end.

(c) Email. Compose an email and send.

(d) Antivirus. Perform security scan.

(e) Launcher. Slide from left to right every 2 seconds for 30 seconds.

(f) Instant messaging and calling. Send five messages to a friend.

(g) Shopping. Search for "socks", scroll the results and finally click on the first one.

(h) File explorer. Scroll the root directory and create a new directory.

Figure 1: Energy consumption of similar apps in 8 app groups. Energy drain numbers (in μAh) are direct output of EPROF, for the actual tests, which vary between 30 seconds to 1 minute long for different app groups.

between 89.3% to 94.9% for File explorer apps. We thus draw our third observation that *(O3) the heavy usage of framework services leads to a high percentage of app energy drain occurring in these shared services, up to over 90% of the app energy consumption.* This phenomenon suggests learning more efficient implementations of app functions by comparing their energy footprints not only is possible, but actually is a meaningful and practical approach.

3 How to Diff Energy Profiles?

The above three key insights suggest comparing the energy profiles of similar apps generated by a source-code energy profiler has the potential to automatically identify inefficiencies in implementing common app functions in similar apps. We call this approach *differential energy profiling*, or *energy diffing* for short.

Developing such a differential energy profiler has to address three challenges: (1) What is the diffing granularity? (2) How to identify the diffing units in the energy profiler output of each app? (3) How to actually diff the energy profiles of similar apps?

3.1 What diffing granularity?

A mobile app typically implements many features. We refer to the implementation of individual app features in the source code and their invocations at runtime as *app tasks*. Similar apps are expected to implement a common set of core tasks pertaining to the apps' common, main functionality, *e.g.,* music playback along with some basic UI features (*e.g.,* progress bar) for music streaming apps.

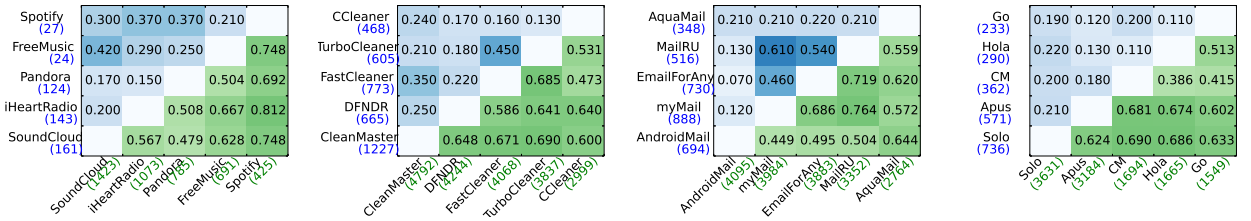
In addition, similar apps by different vendors often support some differentiating features which result in different tasks at runtime. For example, among the five streaming apps, SoundCloud uniquely depicts the audio track using a waveform animation during music playback.

Since there are two potential factors that contribute to the different energy drain of similar apps: (1) different implementation of common app tasks, and (2) app tasks unique to each of the similar apps, the natural granularity for energy diffing of similar apps should be an app task.

3.2 How do app tasks manifest in call trees?

Diffing at the task granularity, however, faces a fundamental challenge: *app tasks are not explicitly labeled by developers*. To overcome the above challenge, we examine how app tasks manifest in the call trees of Android apps.

Android app programming is event-driven where the Android framework implements frequently used tasks as services. These Android framework services provide sev-



(a) Music streaming. Streaming music for 30 seconds.

(b) Cleaning. Press clean and wait for operation to end.

(c) Email. Compose an email and send.

(d) Launcher. Slide from left to right every 2sec for 30 seconds.

Figure 2: Pairwise overlap of similar apps. Lower triangle boxes show the percentage of overlapping framework method calls (0.30 means 30%). Upper triangle boxes show the matched app tasks in percentage of all tasks.

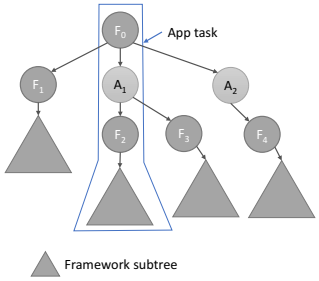


Figure 3: A typical call tree.

eral Java interfaces and classes with callback methods that apps can override. Apps then use the associated registration-callback mechanism to register the overridden callback app methods with the framework. Upon an event, the Android framework calls these overridden methods registered for the event.

We refer to Android framework methods as *F-methods* and app methods as *A-methods*. The above asynchronous programming suggests (1) an app's energy profiling output typically consists of many call trees [9], one for each thread; (2) as shown in Figure 3, each call tree typically starts with some framework method (F_0) that receives call-back related messages and makes a callback into the app (A_1). The app callback method (A_1) may call various other app methods (folded in A_1) which later call another framework method (F_2 or F_3) to register more callbacks (A_2) or for general processing that implements the task.

Using a script, we examined the call tree output by EPROF for all the apps in Figure 1 and confirmed that their call trees all follow the above structure, with one minor variation: a path may contain only F-methods (*e.g.,* (F_0, F_1)). This happens when an app task calls some framework method X that in turn registers an asynchronous callback of some other framework method Y. When framework method Y is invoked, it starts a new path off the root of the call tree consisting entirely of framework methods. Typical general-purpose framework methods that serve as the roots of the call trees include `Handler.dispatchMessage` and `Binder.execTransact`.

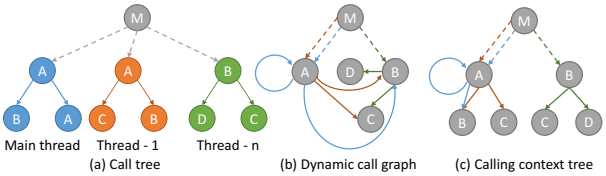


Figure 4: Call trees, dynamic call graphs, and calling context trees.

What constitutes a task in the call tree? The above call tree structure suggests an app task typically manifests in a call tree in an Erlenmeyer flask-shaped [19] slice with three components [2], as shown in Figure 3:

- *Call path:* The call path from the root of the call tree consisting of some F-methods followed by some A-methods that lead to the F-method uniquely captures the context of the task, *i.e.,* under which the F-method was called;
- *F-method:* The specific F-method invoked by the app method that is the entry to the invoked framework service that accomplishes the app task;
- *Subtree:* The actual execution of the F-method, given the context and the parameters passed to the entry method.

We denote the three-component structure as an EFLASK, which is a (path, F-method, subtree) task tuple.

In practice, it is often not obvious to isolate all the EFLASKS in a given call tree that correspond to app tasks, due to the possibly many layers of interleaving of A-methods and F-methods. Our EFLASK matching algorithm described in §3.4 takes the call trees of two similar apps and simultaneously identifies EFLASKS corresponding to app tasks and finds matching tasks.

3.3 What tree structures to diff?

Before discussing the diffing algorithm, we first explore different options of tree structures to perform diffing, as shown in Figure 4.

Call tree Since EPROF outputs a call tree for each execution profile, the baseline approach would be to directly

diff the two call trees (CT). However, this is not practical, since an app task may be invoked many times during a profiling run and thus its task tuple may appear many times in the call tree output. Further, the call tree becomes hopelessly large, up to several million call tree nodes in just a few minutes of a typical profiling run.

Dynamic call graph An alternative approach is to convert call trees to dynamic call graphs (DCG) [9] and diff DCGs instead, where every method executed has just one corresponding method node in a DCG. However, using DCG faces a fundamental challenge that a DCG is not path preserving, *i.e.,* it may contain code paths that never occurred during the profile run. For example, the DCG in Figure 4(b) contains path $M \rightarrow A \rightarrow B \rightarrow D$ which never occurred in the CT in Figure 4(a). Paths need to be preserved for matching the EFLASKS of the same app task.

Calling context tree DIFFPROF overcomes the above shortcomings of CT and DCG by building and using calling context trees (CCT) [9], a middle ground between call trees and dynamic call graphs. In a nutshell, two method call nodes in the call tree are merged in the CCT whenever both nodes have an identical path from the root. In addition, recursive calls are merged to the non-recursive ancestor to keep the tree bounded in size, for example node A in Figure 4(c)[3]. Thus, using a CCT preserves the valuable path information while significantly reducing the number of nodes in the tree. In practice, we found CCT to contain only tens of thousands of nodes in a few minutes of profiling run, allowing our sophisticated matching algorithm to run in less than 30 seconds (§5).

3.4 How to perform EFLASK matching?

We first discuss the need for approximate matching to find EFLASKS corresponding to the same app tasks. We then review prior tree matching algorithms, discuss their drawbacks when applied to our problem, followed by our EFLASK matching algorithm.

3.4.1 Need for approximate DIFFPROF matching

The above understanding of how app tasks manifest in call trees in §3.2 suggests that different implementations and hence their EFLASK structures of the same task in two apps can differ in the following ways:

- *The corresponding call paths may differ slightly.* This can happen for two main reasons. First, apps may use slightly different mechanisms to achieve the same app callback. For example, an app can start its `Runnable.run` method directly from a new thread, or via `ExecutorService`; the two lead to different paths from root. Second, the app can use different app callbacks for receiving similar events. For example, the Turbocleaner app handles the "clean" button press using `.onClick` callback while the DFNDR app uses `.onItemClick` callback after which both apps call `Activity.startActivity` to perform a common task.
- *The entry F-methods may differ* due to two main reasons. First, the same task API can be provided by many different framework classes. For example, both `HttpsURLConnectionImpl.getInputStream` and `HttpURLConnectionImpl.getInputStream` get data from a server, one from an https and another from an http connection. Second, the same framework class may provide many alternate APIs to perform the same app task. For example, three different apps, Wish, Kohl and letgo, share 8 common nodes in the call path from the root call and finally call three different APIs, `ImageView.setImageDrawable`, `ImageView.setImageBitmap` and `ImageView.setImageResource`, respectively, for setting an image.
- *The subtrees that reflect the actual executions of the app task in similar apps can differ.* Even when the developers use the same framework API call to accomplish a task, the program state and the call parameters passed in can differ which lead the framework service to take different paths resulting in different subtrees.

3.4.2 Prior tree matching algorithms

How to match two trees to find similar components has been previously studied with a diverse set of applications such as matching RNA structures, structured text databases and image analysis [12]. However, prior matching algorithms are not suitable for matching EFLASKS.

Exact path matching Let T_1 and T_2 be two CCTs rooted at r_1 and r_2, with the set of nodes denoted by $V(T_1)$ and $V(T_2)$. Formally, exact path matching produces a maximal one-to-one node matching[4] $M \subseteq V(T_1) \times V(T_2)$, where for any pair $(v,w) \in \{M - (r_1,r_2)\}$:

$$(r_1,r_2) \in M \text{ and } (P(v),P(w)) \in M \quad \text{(Path Condition)} \qquad (1)$$

where $P(v)$ and $P(w)$ are parents of nodes v and w respectively. However, exact path matching cannot match paths (*e.g.,* of EFLASK) with minor variations.

Prior approximate tree matching algorithms Tai *et al.* [38] gave the first approximate tree matching algorithm. This algorithm produces a maximal one-to-one matching M where for any pair $(v_1,w_1),(v_2,w_2) \in M$:

$$v_1 \text{ is ancestor of } v_2 \text{ iff } w_1 \text{ is ancestor of } w_2 \quad \text{(Ancestor Condition)} \qquad (2)$$

The output matching replaces the Path Condition in Eqn. 1 with a significantly weaker *Ancestor Condition* (*i.e.,* Path Condition implies Ancestor Condition). However, the algorithm is Max-SNP hard.

To reduce the running time, Zhang *et al.* [44] added a *Structure Respecting Condition* to output matching. This algorithm produce a matching M, such that for any pairs (v_1,w_1), (v_2,w_2), $(v_3,w_3) \in M$:

$$nca(v_1,v_2) = nca(v_1,v_3) \text{ iff } nca(w_1,w_2) = nca(w_1,w_3)$$
(Structure Respecting Condition) (3)

where $nca(x,y)$ is the nearest common ancestor of nodes x and y. Due to the additional constraint, fewer matching possibilities need to be considered, making the algorithm's running time polynomial.

However, these algorithms may match EFLASKS with very different call paths. In contrast to the exact path matching algorithm which focuses on matching the path (component of EFLASKS) without considering the subtrees underneath, the above approximate matching algorithms match two nodes only based on similarity of subtrees (another component of EFLASKS) underneath them disregarding the call paths. The EFLASK matching algorithm we propose below leverages both the path and subtree information in matching two nodes, and in doing so, matches two EFLASKS.

3.4.3 The EFLASK matching algorithm

The EFLASK matching algorithm relaxes the Path Condition incrementally, *i.e.,* the paths from root to matched nodes in two trees can differ by *at most* α *nodes*, and maximizes the subtree overlap. We replace the Path Condition in the exact matching algorithm with a *Relaxed Path Condition* while retaining the Structure Respecting Condition (Eqn. 3) and Ancestor Condition (Eqn. 2) to find such matching. Formally, we wish to produce a maximal one-to-one matching M, that satisfies Eqn. 2 and Eqn. 3 and for any pair $(v,w) \in M$:

$$w \in C_\alpha(v)$$
(Relaxed Path Condition) (4)

where $C_\alpha(v) \subseteq V(T_2)$ is the *candidate set*, where the path from T_2's root to each node in $C_\alpha(v)$ differs from the path from T_1's root to v by less than or equal to α nodes. For example, Figure 5 highlights the nodes in the candidate set $C_\alpha(b)$ for α equal to 0, 1 and 2. $C_0(b)$ contains just 1 node that has the same path from its root as the b in T_1. $C_1(b)$ includes 3 additional nodes a, b and c whose path from root becomes identical to b's from T_1's root, $r \to a \to b$ by doing exactly one operation – deleting b, deleting a and replacing b by c, respectively.

Notations Before presenting the algorithm we define a few notations. Let T_1, T_2 denote two unordered labeled tree with maximum degrees D_1 and D_2, respectively. We denote the set of children nodes of node v by $child(v)$ and its label by $label(v)$. The path from the root to node v thus forms a string of labels and is represented by $s(v)$.

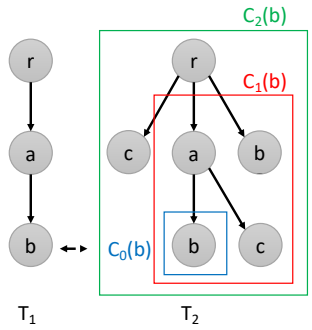


Figure 5: Candidate set $C_\alpha(b)$ for $\alpha = 0$, 1 and 2.

Let θ denotes an empty tree and let $T(v)$ denote the subtree of T rooted at a node $v \in V(T)$ and $F(v)$ denote the forest under node v, $F(v) = T(v) - \{v\}$.

While matching the nodes in two trees, we can perform three types of edit operations to the tree nodes – (1) a relabeling operation to change the node label, (2) a deletion operation to delete node v and make all the children of v the children of $P(v)$, and (3) an insertion operation, the complement of deletion.

Let λ denote a special blank symbol. The cost of each edit operation can be specified using a cost function, γ. Thus, $\gamma(l_1,l_2)$ is the cost of replacing l_1 by l_2, $\gamma(l_1,\lambda)$ is the cost of deleting l_1 and $\gamma(\lambda,l_1)$ is the cost of inserting l_1. γ is generally assumed to be a distance metric, *i.e.,* γ is non-negative, symmetric and follows triangular inequality. We extend the notation such that $\gamma(v,w)$ for nodes v and w denotes $\gamma(label(v),label(w))$. We assume unit cost distance in the design of algorithm, *i.e.,* $\gamma(l_1,l_2) = 1$ when $l_1 \neq l_2$.

Now we are ready to define a few functions and their properties which form the basis of our algorithm.

Path edit distance function We first find $C_\alpha(v)$ by computing a *path edit distance function* ρ. For some $v \in V(T_1)$ and $w \in V(T_2)$, $\rho(s(v),s(w))$ is the total cost of edit operations required for v and w to have identical paths from the root. Thus $C_\alpha(v) = \{w \in V(T_2) | \rho(s(v),s(w)) \leq \alpha\}$.

Since paths $s(v)$ and $s(w)$ are strings, path edit distance function $\rho(s(v),s(w))$ is thus equal to the string edit distance [41] between $s(v)$ and $s(w)$ and hence can be calculated in a similar manner.

Since we only care about path edit distance when it is less than or equal to α, we prune some computation as soon as the distance exceeds α. We can show the runtime for computing C_α is $O(min(N_1 D_2^{\alpha+2}, N_1 N_2))$.

Subtree match function Next, we define a *subtree match function* μ_α between two trees. For $v \in V(T_1)$ and $w \in V(T_2)$, $\mu_\alpha(T_1(v),T_2(w))$ is the size of maximal matching of subtrees $T_1(v)$ and $T_2(w)$ where the matching nodes' paths differ by at most α.

Before providing the next lemma, we need the following definition. A restricted matching $RM(v,w)$ is a matching between nodes of $F_1(v)$ and $F_2(w)$ and is defined as follows: (1) $RM(v,w)$ follows all the matching conditions – Relaxed Path Condition (Eqn. 4), Structure Respecting Condition (Eqn. 3), Ancestor Condition (Eqn. 2), and (2) if (p,q) is in $RM(v,w)$, p is in $T_1(v_i)$ and q is in $T_2(w_j)$, then for any (p',q') in $RM(v,w)$, p' is in $T_1(v_i)$ iff q' is in $T_2(w_j)$ where $v_i \in child(v)$ and $w_j \in child(w)$. In other words, node from a subtree $T_1(v_i)$ must only map to nodes of one subtree $T_2(w_j)$ and vice versa.

Motivated by the constrained edit distance algorithm [44], we derive the recurrence relationship for μ_α.

Lemma 3.1. *For all $v \in V(T_1)$ and $w \in V(T_2)$,*

$$\mu_\alpha(T_1(v),\theta) = 0$$
$$\mu_\alpha(\theta,T_2(w)) = 0$$
$$\mu_\alpha(T_1(v),T_2(w)) = 0 \qquad \text{if } w \notin C_\alpha(v)$$
$$\mu_\alpha(T_1(v),T_2(w)) = max \begin{pmatrix} \max\limits_{w_j \in child(w)} \mu_\alpha(T_1(v),T_2(w_j)) \\ \max\limits_{v_i \in child(v)} \mu_\alpha(T_1(v_i),T_2(w)) \\ \max\limits_{RM(v,w)} \mu_\alpha(RM(v,w)) \\ \qquad +(1-\gamma(v,w)) \end{pmatrix}; \quad \text{otherwise}$$

Proof. Proof is similar to [44], we skip the details here. □

Again, for any $v \in V(T_1)$, we need to compute the $\mu_\alpha(T_1(v),T_2(w))$ function described above for all $w \in C_\alpha(v)$. The runtime for computing μ_α is $O(N_1 \cdot min(D_2^{\alpha+1},N_2) \cdot (D_1+D_2) \cdot log(D_1+D_2))$[2].

The EFLASK matching algorithm Putting things together, the flexible tree matching algorithm makes two passes. First, it makes a *top-down pass* to compute $C_\alpha(v)$ for all $v \in V(T_1)$, *i.e.,* find nodes with call paths different by at most α nodes. Next, it makes a *bottom-up pass* to compute $\mu_\alpha(T_1,T_2)$. Third, it uses a simple backtracking mechanism to find for each node $v \in T_1$ the matching node $w \in T_2$ that maximizes the T_1-T_2 tree match. Finally, it finds the matching EFLASKS based on these maximally matched nodes.

The two passes together simultaneously accomplish matching of both the call path and the subtree components of similar EFLASKS.

3.5 Preprocessing CCTs to facilitate effective matching

The α value affects the tradeoff between finding more matching tasks (that vary in their call paths) and false positive matches. To make the algorithm more effective, we identified several factors that may increase the path distance between the paths for the same app task, and preprocess the CCTs to remove such factors so that more matchings can be found with smaller α values.

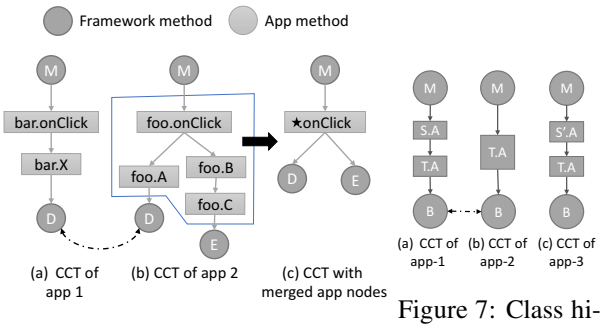


Figure 6: App namespace problem.

Figure 7: Class hierarchy problem.

App namespace problem The call paths for the same task in two CCTs can contain many app methods that are unique to either app as different developers are likely to structure and name the app methods differently. Such app-specific app methods can easily blow up the path edit distance of the call paths of a matching task. Figure 6(a,b) show an example of two paths with differing app methods.

We observe that all the callback app methods must override some predefined framework methods, and the remaining internal app methods called from other app methods have arbitrary names and are also often obfuscated. We thus merge all the internal app method calls into the app callback method root node as shown in Figure 6(c), and drop the app specific class names from app's callback node to allow matching callback methods.

We note that like using DCGs, merging app methods to address the app namespace problem conceptually also reduces path sensitivity, but it actually improves the effectiveness of task matching. This is because the internal methods of different apps tend to be named very differently and thus path sensitivity to app method names actually harms path similarity matching.

Class hierarchy problem A similar issue arises due to the object-oriented nature of Java, as shown in the following example. The two apps in Figure 7(a,b) share a same task pointed by the dashed arrow, but the first app uses method `S.A` which extends and calls method `T.A` and the second app directly uses `T.A`. Each such occurrence in the path increases the path edit distance by one, and more occurrences will quickly inflate the path edit distance.

We solve this problem in two steps. First, we merge `T.A` into the caller node `S.A` (`S'.A`). Second, we tweak the distance function γ to allow matching `S.A` with `T.A`, *i.e.,* $\gamma(\texttt{S.A},\texttt{T.A}) = 0$. This allows matching the common task in Figure 7(a,b) with a path distance of zero while retaining the same path edit distance for sibling classes in Figure 7(a,c).

F-method only paths A third situation happens when a path off the root consists entirely of framework methods as discussed in §3.2 (path (F_0, F_1) in Figure 3). When this happens, in energy profiling, the energy consumption of the call path is not propagated to its asynchronous caller, *i.e.,* the app task, which leaves the developer clueless as to what app task caused the energy drain.

DIFFPROF patches such asynchronous framework only subtrees to its parent app task by adding additional logging in the Android framework. In particular, it logs the callback object's `.hashCode()` along with the current timestamp and thread id, when an asynchronous callback is enqueued in framework and when the callback is later dispatched. During post-processing, for each dispatch method call, the nearest preceding enqueue method call with matching object `.hashCode` log is patched as the dispatch method call's asynchronous caller. [5]

4 Implementation and Usage

We implemented DIFFPROF on top of EPROF [32] with 5.7K lines of Java code. DIFFPROF is packaged as an IDE plugin that can be installed on a laptop, with a GUI front-end, for interacting with the developer and computing and showing the energy diffing result. EPROF traces are collected on a phone running a modified Android 6.0.1 framework version that adds 95 lines to capture hidden causal relationships due to asynchronous programming (§3.5). [6]

After collecting EPROF traces of two similar apps, the developer specifies these traces to DIFFPROF, and DIFFPROF performs energy diffing in the following steps. (1) First, DIFFPROF patches the call tree dumped by EPROF using the call timing and the log timestamp as described in §3.5. (2) Next, DIFFPROF converts EPROF's CT output into CCT and dumps the CCT along with the inclusive and exclusive energy consumption by and the number of recursive and non-recursive invocations of each CCT node. (3) Next, the developer is presented with a list of Java package names that appeared in either app trace to determine app packages used for merging app methods as described in §3.5. By default, all packages not belonging to the Android framework are marked as app packages. For comparing two different apps, developers can skip this step, since packages not belonging to the Android framework are already marked as app packages. When comparing two versions of the same app, however, this presents an opportunity for the developer to unmark certain app packages to expose app-internal path information (Figure 6) during matching. (4) DIFFPROF performs the EFLASK matching algorithm on the pair of CCTs. (5) Finally, since the EFLASKS of multiple tasks may share a common path, DIFFPROF assigns the energy drain for each task as the inclusive energy of the F-method.

Table 2: Average running time and matched tasks when adjusting α. The results are averaged over all app pairs in each group. α=0 gives the exact matching algorithm.

α	0	1	2	3	4	5
Avg. time (sec)	0.20	1.12	4.89	7.79	16.4	25.7
Avg. % of matched tasks	10.8	15.6	18.0	19.5	21.3	22.9

DIFFPROF gives two outputs: (1) a merged list of matched (with the other app) and unmatched tasks in the app, sorted by the energy drain for unmatched tasks and the energy difference for matched ones, *i.e.,* based on the potential room for improvement; and (2) upon selection, a task's EFLASK in a graphical view.

5 Evaluation

Our evaluation answers the following questions: (1) Does DIFFPROF effectively identify matching and unique tasks among similar apps? (2) Does DIFFPROF offer added benefits over EPROF, in particular, how does it help developers with understanding and coming up with more efficient implementation?

5.1 Experimental setup

We use DIFFPROF to profile popular apps belonging to the 8 app groups in Table 1. For each group, we pick 5 different apps and 4 older versions of one of the 5 apps, same as in Figure 1. In running the tests, we ensure user interaction homogeneity using automated testing as described in §2.2. All app tests are less than 1 minute long and are run on a Nexus 6 phone running DIFFPROF's modified version of Android 6.0.1. The traces are post processed and task matching is performed on a Macbook pro laptop with a 2.5 GHz Intel i5 CPU and 8GB 1600 MHz DDR3 main memory.

Impact of α We first evaluate the impact of changing α on the EFLASK matching algorithm's running time and output. Table 2 summarizes the results. We see that as expected, the running time grows close to exponentially with the α value (from 0 to 2 and from 2 to 4). On average, the algorithm produces the energy diffing output within half a minute for all values of $\alpha \leq 5$.

Next, we observe that the average percentage of matching tasks grows steadily as we increase the value of α, starting 10.8% on average at α=0 up to 22.9% at α=5. The growth slows down at $\alpha = 5$.

Based on the above result, when profiling the 8 app groups, for each app pair in a category, we run DIFFPROF to find the matching tasks using the lowest α that can match 20% of the tasks, up to $\alpha = 5$ (shown as dynamic α in Table 3).

5.2 Diffing results

The pairwise task overlap for 4 app groups (Music streaming, Cleaning, Email, Launcher) are shown in the upper triangles in Figure 2. We see that the task overlap between similar apps is significant, ranging between 7%–61%, with an average of 27%, 24%, 28%, and 17%, for the 4 groups, respectively.

Table 3 gives the details of diffing results for each app in the 8 app groups. For each app, we classify all its tasks into tasks that could not be matched with any of the 4 other apps in its category and tasks that were matched with 4, 3, 2 or 1 other app(s). The columns under "Dynamic α" show that the count of such tasks for each app varies for different categories, *e.g.,* Email apps have 17 5-way matching tasks while Music apps have only 2, suggesting the apps in different categories have different levels of overlapping tasks. We manually examined 20% of the matched tasks and did not find any false positives.

Table 3 also shows that the percentage of energy drained by matched tasks (*i.e.,* 1 minus that of unique tasks energy) is over 70% of the total energy drained by the app for 32 out of the 40 apps. This suggests that although it is hard to measure the coverage (false negative) of task matching produced by DIFFPROF, in practice, DIFFPROF produces matched tasks that already account for a majority of the app energy drain which gives app developers enough focus for optimization.

DIFFPROF also exposes app unique tasks that drain significant amounts of energy. Table 3 shows SoundCloud and CM launcher drain 53.7% and 43.7% of the total energy in performing unique app tasks/features, waveform animation and rotation animation, respectively.

To show the effectiveness of the EFLASK algorithm, Table 3 last column lists the number of tasks in each app that do not get matched using the exact path matching algorithm ($\alpha = 0$). We see that the EFLASK matching algorithm with dynamic α reduces the number of unmatched tasks by 13.5% on average (shown in second column).

Table 3: Task overlap for all apps.

	Dynamic α						α=0
App	**0**	**1**	**2**	**3**	**4**	**Unique tasks' energy**	**0**
Antivirus							
AVG	424	191	45	5	5	7.27%	498
CMSecurity	433	169	36	6	5	20.36%	532
DU	252	68	27	8	5	14.10%	286
Kaspersky	126	48	28	7	5	26.01%	149
MobileSec	165	52	39	9	5	30.02%	227
Cleaner							
CCleaner	301	100	41	18	8	23.43%	395
CM	797	290	92	44	8	29.18%	863
DFNDR	402	138	77	46	8	26.48%	495
Fast	265	356	92	55	8	5.11%	286
Turbo	250	234	69	46	8	11.54%	259
Email							
Android Mail	581	67	25	11	17	17.26%	656
Aqua Mail	223	59	28	21	17	6.67%	308
Email For Any	331	154	193	40	17	3.79%	338
Mail RU	131	129	199	47	17	0.60%	145
myMail	434	200	202	40	17	3.10%	454
File Explorer							
ES	244	43	14	4	5	25.15%	272
FX	83	33	5	2	5	4.97%	97
File Exp.	110	42	13	1	5	10.92%	130
File Man.	332	51	9	4	5	24.76%	366
Solid	260	47	16	2	5	7.31%	295
Instant Messaging and Calling							
Hangout	780	160	44	7	8	36.50%	881
Line	291	88	35	21	8	29.14%	411
Messenger	928	256	59	13	8	28.00%	1167
TextNow	1405	194	40	4	8	38.47%	1542
Whatsapp	274	107	26	15	8	40.51%	391
Launcher							
Apus	430	111	23	6	8	32.77%	495
CM	252	65	30	11	8	43.65%	318
Go	161	50	11	8	8	26.21%	204
Hola	212	45	21	4	8	29.74%	252
Solo	560	132	31	7	8	26.23%	640
Music							
FreeMusic	11	6	6	1	2	1.38%	12
Pandora	97	18	5	3	2	17.34%	107
SoundCloud	123	24	10	3	2	53.72%	135
Spotify	14	5	3	3	2	6.23%	14
iHeartRadio	98	34	8	3	2	8.72%	104
Shopping							
Amazon	1030	135	54	17	10	32.03%	1118
Kohl	900	218	80	24	10	27.84%	1041
Wish	1321	264	94	30	10	23.15%	1473
eBay	715	172	86	32	10	26.69%	840
letgo	618	222	108	31	10	19.55%	729
Average	409	119	51	16	8	21.14%	473

5.3 Effectiveness

We discuss how DIFFPROF offers added benefits over a standard energy profiler through extensive case studies. Our case studies show that DIFFPROF provides developers with actionable diagnosis beyond a standard energy profiler in two ways: (1) DIFFPROF identifies non-essential (unmatched or extra) and known-to-be inefficient (matched) tasks; (2) the EFLASKS of tasks extracted by DIFFPROF further expose the reasons for the more efficient implementation. For convenience, in the following, we often refer to a task by the F-method in its EFLASK 3-tuple.

Methodology We ran DIFFPROF on the top 3 energy-draining apps in each of the 8 groups against the least

Table 4: Buggy and inefficient tasks in case studies and their energy drain.

App	Task	Task energy drain (μAh)	% of total energy drain
Unmatched tasks			
Hangout	ContentResolver.query	44.3	10.1%
Kohl	ObjectInputStream.readObject	12.8	3.9%
Kohl	ObjectOutputStream.writeObject	10.5	3.2%
Kaspersky	Thread.getStackTrace	39.6	14.8%
Pandora 8.0	SharedPreferencesImpl $EditorImpl.apply	22.9	17.5%
DFNDR	Runtime.exec	19.5	5.2%
Matched tasks			
Wish letgo	Bitmap.compress	100.9 7.14	15.9% 3.6%
Wish letgo	BitmapFactory.decodeStream	126.3 5.01	19.9% 2.5%
Pandora5.7 Pandora8.3	TextView.setText	43.6 0.74	28.1% 0.7%
Spotify Pandora	ProgressBar.setProgress	29.2 1.74	20.2% 1.6%
TextNow Whatsapp	ViewRootImpl.performTraversal	230.5 24.0	40.6% 28.4%
Solid FX	Drawable.invalidateSelf	35.5 1.24	18.9% 2.0%

Table 5: Rank ordered EPROF's method energy output and DIFFPROF's task energy difference output for Google Hangout compared to Whatsapp. Energy in μAh. "*": unmatched tasks.

Rank	Method name (EPROF output)	Inclusive energy
1	(toplevel)	436.8
2	void Looper.loop()	220.6
3	void Handler.dispatchMessage(Message)	207.3
4	void Thread.run()	176.9
5	Object Method.invoke()	175.4
27	Cursor ContentResolver.query()	44.3
Rank	Method name (EPROF output)	Exclusive energy
1	boolean BinderProxy.transactNative()	50.8
2	void VMRuntime.runHeapTasks()	11.6
3	void MessageQueue.nativePollOnce()	9.86
4	Object Throwable.nativeFillInStackTrace()	9.39
5	void Trace.nativeTraceBegin()	7.81
1336	Cursor ContentResolver.query()	0.00
Rank	Task name (DIFFPROF output)	Task energy
1	Cursor ContentResolver.query()*	44.3
2	int TelephonyManager.getSimState()*	24.9
3	Cursor SQLiteQueryBuilder.query()*	17.2
4	void ObjectOutputStream.writeObject()	11.2
5	Spanned Html.fromHtml()	6.49

energy-draining app in the same group, and looked at the top energy-draining app tasks output by DIFFPROF. Out of these, we skip the cases where the app tasks are for supporting unique app features (*e.g.,* 47.2% of SoundCloud's total energy was by a task supporting the waveform animation feature). The remaining 12 tasks, summarized in Table 4, all belong to buggy or inefficient implementations, removing which reduces the app energy drain by 5.2%–27.4% (based on the energy difference).

5.3.1 Unmatched (extra) tasks

Instant Messaging Table 5 shows Google Hangout's energy output from EPROF and from DIFFPROF when compared with Whatsapp. When sorted by inclusive energy, EPROF shows really high-level Android methods such as `Looper.loop` on the top, and when sorted by exclusive energy, it shows really low-level Android methods such as `BinderProxy.transactNative` on the top. Such top energy drainers in both inclusive and exclusive energy lists are F-methods that do not directly call app methods and are not directly called by the app; the developers thus do not get useful guidance on what to focus on from the long list of EPROF output.

In contrast, DIFFPROF outputs tasks sorted by energy drain. It shows Hangout consumes more than 10% of its total energy in an unmatched task `ContentResolver.query`. Since tasks' F-methods are directly called by the app, the top task's name provides direct hints to developer on how to optimize the app. EPROF, however, does not highlight such methods; the top task method appeared at position 27 when sorted by inclusive energy and at 1336 when sorted by exclusive energy.

Finding the reasons and optimization for task `ContentResolver.query` would have been easy for its developer from the EFLASK output, *e.g.,* the `ContentResolver.query` method was called 116 times. But since we did not write the app, to understand this energy drain, we set a breakpoint at the `ContentResolver.query` method and reran the app to examine the parameters passed to the method. In one call to the method, the app queries multiple fields that are stored in a local database. We found that at one message send, the app queries for 81 unique database fields which often are repeated across two different queries. Moreover, 36 out of the 81 fields, such as `author_chat_id` and `author_first_name`, do not change across two send key presses, but keep on getting queried at each send. This suggests that there is ample room for optimization by keeping a staleness flag; only when the user navigates away from a chat window, the 36 fields can be declared stale and re-queried later.

Shopping Table 6 shows the Kohl's app's output from EPROF and from DIFFPROF when compared with letgo. DIFFPROF shows `ObjectInputStream.readObject` and `ObjectOutputStream.writeObject` are two top energy draining extra tasks, consuming 3.9% and 3.2% respectively of its total energy consumption. In contrast, EPROF outputs them at positions 90 and 133 when sorted by inclusive energy and at 1516 and 1547 when sorted by exclusive energy, respectively.

Table 6: Rank ordered EPROF's method energy output and DIFFPROF's task energy difference output for Kohl compared to letgo. Energy in μAh. "*": unmatched tasks.

Rank	Method name (EPROF output)	Inclusive energy
1	(toplevel)	329.71
2	Object Method.invoke()	149.74
3	void Looper.loop()	134.65
4	void ActivityThread.main()	132.89
5	void ZygoteInit$MethodAndArgsCaller.run()	132.89
90	Object ObjectInputStream.readObject()	12.82
133	void ObjectOutputStream.writeObject()	10.53
Rank	Method name (EPROF output)	Exclusive energy
1	void VMRuntime.runHeapTasks()	26.11
2	boolean BinderProxy.transactNative()	11.9
3	Bitmap BitmapFactory.nativeDecodeByteArray()	10.34
4	void DdmVmInternal.threadNotify()	10.02
5	String StringFactory.newStringFromChars()	7.96
1516	Object ObjectInputStream.readObject( )	0.0
1547	void ObjectOutputStream.writeObject()	0.0
Rank	Task name (DIFFPROF output)	Task energy
1	Object ObjectInputStream.readObject()*	12.82
2	Bitmap BitmapFactory.decodeByteArray()	11.19
3	void ObjectOutputStream.writeObject()*	10.53
4	boolean Class.isAnonymousClass()	9.55
5	String JSONObject.toString()	8.24

Since we did not write the app, we dug into the energy drain by setting breakpoints. We found that the app keeps the entire catalog and current discount campaigns on the SD card in `catalog.tmp` and `cms.tmp` files respectively which were 227 KB and 21 KB at the time of the experiment. Whenever a new catalog or a new campaign is synced with the server, the entire files are dumped again, rewriting the previous entries; using a database just to update new entries would have been more efficient.

Note that task `View.draw` consumes 12.16 μAh energy, more than the above extra tasks, but does not appear in the top task list. This is because DIFFPROF prioritizes the tasks with the most room for optimization: since the letgo app consumes 8.29 μAh for the same task, the difference is less than 4 μAh.

Antivirus DIFFPROF highlights `Thread.getStackTrace` as an extra task in the Kaspersky app which consumes 39.57 μAh, 14.8% of the app's total energy drain (position 1 in DIFFPROF output, but 22 in EPROF output). After decompiling the app apk using dex2jar [4], we inspected the caller of `Thread.getStackTrace` in the app source code and found that the app collects logs with unicode characters but in every such attempt, the code throws `UnsupportedEncodingException` which internally collects the thread stack trace thus unnecessarily wasting energy. This bug was confirmed by Kaspersky developers.

Music DIFFPROF highlights `SharedPreferencesImpl$Editor.apply` as an extra task in Pandora v8.0 that consumes 17.5% of its total energy drain (position 4 in DIFFPROF output but 42 in EPROF output). This method is used to change app preferences. The Android developer manual suggests that apps should call `SharedPreferencesImpl$Editor.edit` repeatedly to keep making changes in memory and then call `SharedPreferencesImpl$Editor.apply` once at the end to commit all the changes to the disk. However, the app mistakenly calls `SharedPreferencesImpl$Editor.apply` once every second. This bug was confirmed and fixed in the latest version of the Pandora app.

Cleaner DIFFPROF shows that the DFNDR app calls framework method `Runtime.exec`, consuming 19.52 μAh, 5.2% of the app's total energy consumption (position 3 in DIFFPROF output but 50 in EPROF output). We set a breakpoint at this method and examined its parameters and found that the app runs `ps | grep <app_pkg>` for each app installed on the phone. Since `ps` walks down the entire `/proc` directory, it would be more efficient to just obtain the `ps` output once and parse it to find the fields related to each app.

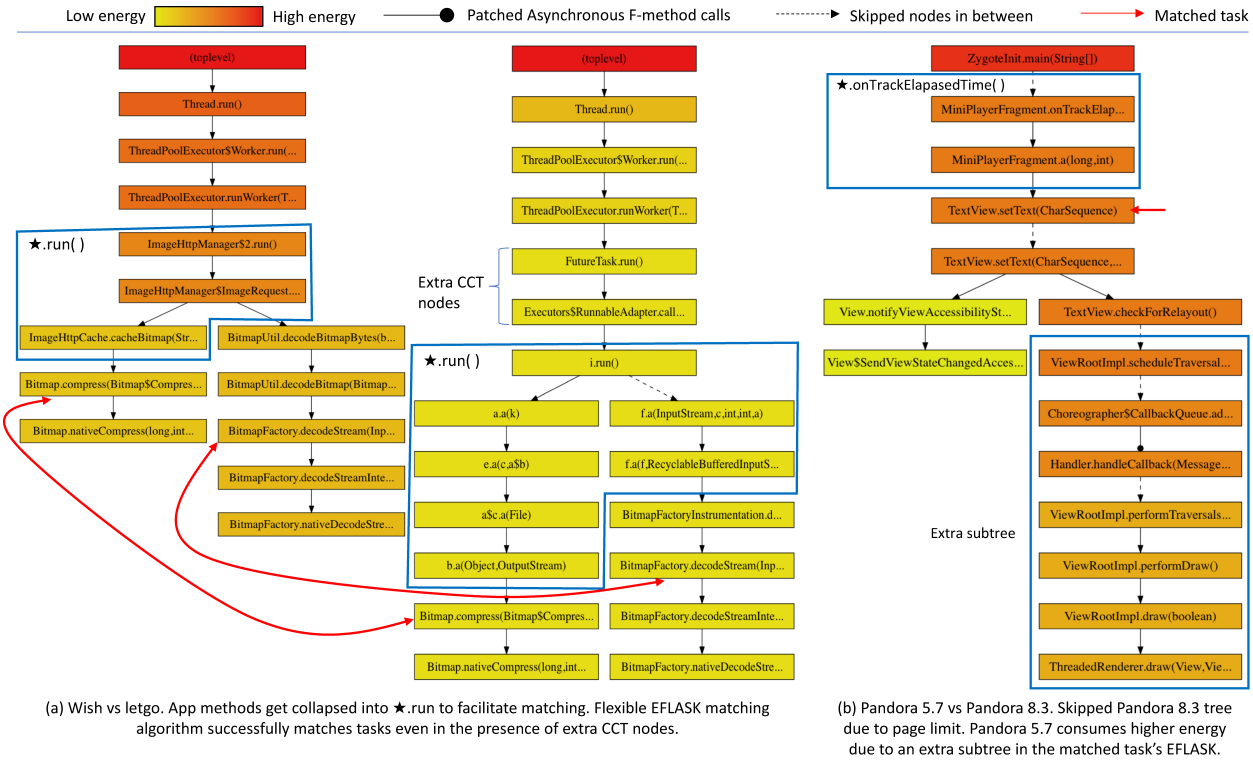
5.3.2 Matched tasks

Shopping In diffing Wish and letgo, although the CCTs of the two apps differ a lot structurally as shown in Figure 8(a), DIFFPROF is able to match two commons tasks, `Bitmap.compress` and `BitmapFactory.decodeStream`, by collapsing app methods to `⋆.run` and its flexible EFLASK matching algorithm.

For the `Bitmap.compress` task, DIFFPROF shows that Wish consumes 100.94 μAh, 15.9% of its total energy drain whereas letgo consumes only 7.14 μAh. To find the root cause of energy difference, we examined the parameters passed to the F-method by setting a breakpoint and rerunning both apps. We found that Wish compresses the image into a png image with quality set to 100 while letgo compresses into a jpg image with quality set to 90. This causes the large energy difference while the images shown by both apps are visually similar.

The above image format difference also explains the energy drain difference between the second common task `BitmapFactory.decodeStream` where Wish consumes 126.32 μAh, 19.9% of its total energy drain while letgo consumes only 5.01 μAh.

Music – Pandora In diffing two versions of Pandora, DIFFPROF matches the common task `TextView.setText` even though structurally their EFLASKS look different, as shown in Figure 8(b) (merged to save space). DIFFPROF shows that the common task consumes 43.63 μAh, 28.1% of its total energy consumption in Pandora v5.7 but only 0.74 μAh in the latest Pandora app, v8.3.



(a) Wish vs letgo. App methods get collapsed into ★.run to facilitate matching. Flexible EFLASK matching algorithm successfully matches tasks even in the presence of extra CCT nodes.

(b) Pandora 5.7 vs Pandora 8.3. Skipped Pandora 8.3 tree due to page limit. Pandora 5.7 consumes higher energy due to an extra subtree in the matched task's EFLASK.

Figure 8: Matched tasks between (a) Wish and letgo, (b) Pandora v5.7 and v8.3.

DIFFPROF further highlights the reason for the difference: in Pandora v5.7, the subtree additionally contains the `ViewRootImpl.scheduleTraversal` subtree that traverses and measures the entire view hierarchy. We used a premium account to disable ads and played the same radio station on both Pandora versions for two hours while leaving the phone on the playback screen. We found that Pandora v5.7 drained 9.2% battery per hour whereas Pandora v8.3 drained only 6.7% battery per hour. We reported this bug to Pandora engineers, who verified that Pandora v5.7's layout.xml file erroneously declared the width of elapsed time and remaining time text views to `wrap_content`. This flag signals Android's `ViewManager` that the text view must be just large enough to enclose its content. As a result, every second when the app updates the elapsed time and remaining time text views, Android `ViewManager` traverses the entire view hierarchy to recompute the size of the text boxes. The text boxes were set to a fixed size in later versions of Pandora.

Music – Spotify In diffing Pandora and Spotify apps, DIFFPROF shows that the common `ProgressBar.setProgress` task consumes 43.63 μAh, 28.1% of its total energy in Spotify, but just 1.74 μAh in Pandora. The EFLASK output further shows that Spotify calls this method from `App.doFrame` 596 times while Pandora calls it only 29 times from `App.onTrackElapsedTime` during the 30 second music playback, *i.e.,* while Pandora updates the progress bar once per second, Spotify updates it on every frame, which is unnecessarily frequent as many frame draws lead to no pixel change.

Instant Messaging In diffing TextNow and Whatsapp, DIFFPROF shows that TextNow consumes 230.46 μAh, 40.6% of its total energy drain, in calling a common task `ViewRootImpl.performTraversal`, almost 10 times that in Whatsapp. On inspecting the layout of the two apps with Android's `HierarchyViewer`, we found that TextNow contains 226 views compared to 76 in Whatsapp. Our closer inspection of view properties shows that 172 views in TextNow are in fact not even visible on the screen. The app statically loads all the possible UI interactions such as `pause_playing_voice_note_button` and `change_billing_details_button_icon`, keeping them all in the view hierarchy instead of dynamically loading views on demand as recommended by Android [3] and thus inflating the view hierarchy traversal energy. Moreover, the app contains several `LinearLayout` with just an `ImageView` and a `TextView` which are recommended to be compressed into one compound view [8] to reduce the size of the view hierarchy.

File Explorer DIFFPROF shows that Solid explorer consumes 35.52 μAh, 18.9% of its total energy in task `Drawable.invalidateSelf` whereas FX file explorer only consumes 1.24 μAh. DIFFPROF further shows that Solid calls `Drawable.invalidateSelf` 1002 more

times than FX and that the EFLASK contains `ObjectAnimator.animateValue` followed by Solid's `CircularAnimatedDrawable$1.set`. Upon inspecting this class, we found that the app does the animation when a new folder is created. At each frame, it draws an arc and requests another frame. However, after the folder gets created, the app stops drawing the arc but keeps requesting new frames, unnecessarily wasting energy.

6 Discussions

DIFFPROF's effectiveness in finding energy optimizations stems from the large overlap of Android libraries used among competing Android apps and accurate source-level energy profiling. As such, its central idea of diffing source-code-level profiling of similar apps in principle can be extended to find optimization opportunities in other performance metrics of interests to developers, such as latency, scalability and memory efficiency.

One of the central principles of software engineering, DRY (Don't repeat yourself) [24], preaches the use of reusable code, by abstracting all common reusable code into standalone libraries. The principle improves modern software developers' productivity and has gained wide adoption in recent years; almost every major build tool today [1, 6, 2] allows developers to specify library dependencies which are downloaded from a central repository and packaged with their software. We envision that DIFFPROF's approach can be extended to effectively compare source-code level profiling measurements of software in broader domains beyond mobile such as games, web frontends and server backends.

7 Related work

Performance and energy profiling There is a large body of work on performance profiling of sequential programs [20, 15, 30] and concurrent programs [17, 39]. There are also several works on energy profiling for mobile apps [32, 31, 34, 18]. EPROF [32] performs source-code-level energy profiling and accounts the energy drained by each phone component to individual app method calls. ARO [34] performs cross-layer profiling for network usage to expose apps' inefficient interactions with lower layers. Wattson [31] estimates app energy consumption on the developer workstation by emulating different environments such as network conditions, CPU speed and display technologies. GfxDoctor [18] quantifies the energy drain spent in traversing the entire frame rendering stack due to each UI update. All such profilers stop at finding performance/energy hotspots. DIFFPROF builds on top of such traditional profilers and tackles the hard but critical question in the app energy optimization process: whether and how energy hotspots in app source code can be restructured to drain less energy.

Diffing programs and runtime behavior. (1) **Programs.** There has been a large body of research to find regressions introduced from code revisions [13, 36, 22, 23], and on data mining application source code to detect software bugs, *e.g.,* [40]. DIFFPROF allows app developers to catch and debug energy drain regressions by comparing source-code energy profiles after code revisions. (2) **Runtime behavior.** Execution indexing [43] aligns event logs of two executions of the same program under different input or perturbations and has been used in detecting and understanding security leaks [27], deadlocks [28] and failures [45, 21]. DIFFPROF aligns calling context trees of two executions that may be from apps written by different developers to find energy inefficiencies.

Diffing beyond programs. More generally, diffing is a pervasive technique that celebrates and exploits diversity and has been applied to many other scenarios in computer systems and networking. Diffing data has been applied to storage data for data compression (*e.g.,* [29]), to network traffic for traffic reduction (*e.g.,* [11, 10]), to data structures in memory images for detecting polymorphic malware [16], and to frames for reducing graphics energy for mobile devices [25].

Beyond data, many systems, *e.g.,* PeerPressure [42], ClearView [33], Shen et al. [37], Encore [46], and DiffProv [14], apply diffing to learn or detect deviations from the correct or reference behavior, via statistical analysis or data mining, for detecting and diagnosing misconfigurations, performance anomalies or faulty events in the network and distributed systems.

8 Conclusion

This paper presents differential energy profiling which tackles the hard but critical question in the app energy optimization process faced by app developers: whether and how energy hotspots in app source code can be restructured to drain less energy. By performing approximate matching of energy profiles of similar apps by a traditional energy profiler, energy diffing automatically uncovers more efficient implementations of common app tasks and app-unique tasks among similar apps. We show how our prototype DIFFPROF tool provides developers with actionable diagnosis beyond a traditional energy profiler: it effortlessly reveals 12 inefficient or buggy implementations in 9 apps, and it further allows (non)developers to quickly understand the reasons and develop fixes for the energy difference.

Acknowledgement We thank our shepherd Andreas Haeberlen and the anonymous reviewers for their helpful comments which helped to improve this paper. This work was supported in part by NSF grant CSR-1718854.

References

[1] Apache maven project. http://maven.apache.org.

[2] Create .net apps faster with NuGeT. https://www.nuget.org.

[3] Delayed loading of views. https://developer.android.com/training/improving-layouts/loading-ondemand.html#ViewStub.

[4] dex2jar. https://sourceforge.net/projects/dex2jar/.

[5] dexinfo. https://github.com/poliva/dexinfo.

[6] Npm package manager. https://www.npmjs.com.

[7] Optimizing battery life. https://developer.android.com/training/monitoring-device-state/index.html.

[8] Using compound drawables. https://developer.android.com/training/improving-layouts/optimizing-layout.html#Lint.

[9] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices 32*, 5 (1997), 85–96.

[10] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. of ACM SIGCOMM* (2008), pp. 219–230.

[11] ANAND, A., SEKAR, V., AND AKELLA, A. Smartre: an architecture for coordinated network-wide redundancy elimination. In *Proc. of ACM SIGCOMM* (2009), pp. 87–98.

[12] BILLE, P. A survey on tree edit distance and related problems. *Theoretical computer science 337*, 1 (2005), 217–239.

[13] CALCAGNO, C., DISTEFANO, D., DUBREIL, J., GABI, D., HOOIMEIJER, P., LUCA, M., OHEARN, P., PAPAKONSTANTINOU, I., PURBRICK, J., AND RODRIGUEZ, D. Moving fast with software verification. In *NASA Formal Methods Symposium* (2015), Springer, pp. 3–11.

[14] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. of ACM SIGCOMM* (2016), pp. 115–128.

[15] COPPA, E., DEMETRESCU, C., AND FINOCCHI, I. Input-sensitive profiling. *ACM SIGPLAN Notices 47*, 6 (2012), 89–98.

[16] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proc. of USENIX OSDI* (2008), pp. 255–266.

[17] CURTSINGER, C., AND BERGER, E. D. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 184–197.

[18] DING, N., AND HU, Y. C. Gfxdoctor: A holistic graphics energy profiler for mobile devices. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 359–373.

[19] Erlenmeyer flask. https://en.wikipedia.org/wiki/Erlenmeyer_flask.

[20] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. gprof: A call graph execution profiler. In *Proc. of ACM PLDI* (1982).

[21] GUO, L., ROYCHOUDHURY, A., AND WANG, T. Accurately choosing execution runs for software fault localization. In *International Conference on Compiler Construction* (2006), Springer, pp. 80–95.

[22] GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. An approach to regression testing using slicing. In *Software Maintenance, 1992. Proceedings., Conference on* (1992), IEEE, pp. 299–308.

[23] HASSAN, A. E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 78–88.

[24] HUNT, A., AND THOMAS, D. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley Longman, Inc., ISBN-10: 020161622X., 1999.

[25] HWANG, C., PUSHP, S., KOH, C., YOON, J., LIU, Y., CHOI, S., AND SONG, J. Raven: Perception-aware optimization of power consumption for mobile games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (2017), ACM, pp. 422–434.

[26] Energy efficiency and the user experience. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/EnergyandNetworking.html.

[27] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential slicing: Identifying causal execution differences for security applications. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 347–362.

[28] JOSHI, P., PARK, C.-S., SEN, K., AND NAIK, M. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. of ACM PLDI* (2009), pp. 110–120.

[29] KULKARNI, P., DOUGLIS, F., LAVOIE, J. D., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track* (2004), pp. 59–72.

[30] KÜSTNER, T., WEIDENDORFER, J., AND WEINZIERL, T. Argument controlled profiling. In *European Conference on Parallel Processing* (2009), Springer, pp. 177–184.

[31] MITTAL, R., KANSAL, A., AND CHANDRA, R. Empowering developers to estimate app energy consumption. In *Proc. of ACM MobiCom* (2012).

[32] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys* (2012).

[33] PERKINSA, J. H., KIMB, S., LARSENG, S., AMARASINGHEA, S., BACHRACHA, J., CARBINA, M., PACHECOD, C., SHERWOOD, F., SIDIROGLOUA, S., SULLIVANE, G., WONGZ, W.-F., ERNSTQ, Y. Z. M. D., AND RINARD, M. Automatically patching errors in deployed software. In *SOSP* (2009), pp. 87–102.

[34] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. of Mobisys* (2011).

[35] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. Appinsight: mobile app performance monitoring in the wild. In *Proc. of USENIX OSDI* (2012), pp. 107–120.

[36] ROTHERMEL, G., AND HARROLD, M. J. Selecting regression tests for object-oriented software. In *ICSM* (1994), vol. 94, pp. 14–25.

[37] SHEN, K., STEWART, C., LI, C., AND LI, X. Reference-driven performance anomaly identification. In *ACM SIGMETRICS* (2009), pp. 85–96.

[38] TAI, K.-C. The tree-to-tree correction problem. *Journal of the ACM (JACM) 26*, 3 (1979), 422–433.

[39] TALLENT, N. R., AND MELLOR-CRUMMEY, J. M. Effective performance measurement and analysis of multithreaded applications. In *PPoPP* (2009), ACM, pp. 229–240.

[40] TAN, L., YUAN, D., KRISHNA, G., AND ZHOU, Y. *icomment: bugs or bad comments?*. In *Proc. of ACM SOSP* (2007), pp. 145–158.

[41] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *Journal of the ACM (JACM) 21*, 1 (1974), 168–173.

[42] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with peerpressure. In *Proc. of USENIX OSDI* (2004), pp. 245–258.

[43] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Proc. of ACM PLDI* (2008), ACM, pp. 238–248.

[44] ZHANG, K. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern recognition 28*, 3 (1995), 463–474.

[45] ZHANG, X., TALLAM, S., GUPTA, N., AND GUPTA, R. Towards locating execution omission errors. In *Proc. of ACM PLDI* (2007), ACM, pp. 415–424.

[46] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. Encore: exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS* (2014), pp. 687–700.

Notes

[1]We did not include social networks because their main app functions appear to differ (*e.g.,* Facebook, twitter, snapchat).

[2]F-method-only paths will be patched to other tasks as discussed in §3.5.

[3]Refer to [9] for more details on calling context tree construction.

[4]A maximal one-to-one matching matches the most nodes in the two trees.

[5]Our approach to tracking events is similar to AppInsight [35], but instead of instrumenting app binary, we directly modify the Android framework to track asynchronous calls. Since we use timestamp and thread id in addition to hashCode to track objects, we did not see problems due to hashCode collisions in our experiments.

[6] Since EPROF does not break down app energy drain into native code methods - it simply folds native code's energy into JNI boundary method for Java, DIFFPROF would not be able to identify tasks in native code. In practice, tasks typically start from framework callback Java methods and hence most of the task structures are captured in the Java methods that invoke the native code.

wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events

Fang Zhou, Yifan Gan, Sixiang Ma, Yang Wang
The Ohio State University

Abstract

This paper tries to identify waiting events that limit the maximal throughput of a multi-threaded application. To achieve this goal, we not only need to understand an event's impact on threads waiting for this event (i.e., local impact), but also need to understand whether its impact can reach other threads that are involved in request processing (i.e., global impact).

To address these challenges, wPerf computes the local impact of a waiting event with a technique called *cascaded re-distribution*; more importantly, wPerf builds a *wait-for graph* to compute whether such impact can indirectly reach other threads. By combining these two techniques, wPerf essentially tries to identify events with large impacts on all threads.

We apply wPerf to a number of open-source multi-threaded applications. By following the guide of wPerf, we are able to improve their throughput by up to 4.83×. The overhead of recording waiting events at runtime is about 5.1% on average.

1 Introduction

This paper proposes wPerf, a generic off-CPU analysis method to identify critical waiting events that limit the maximal throughput of multi-threaded applications.

Developers often need to identify the bottlenecks of their applications to improve their throughput. For a single-threaded application, one can identify its bottleneck by looking for the piece of code that takes the most time to execute, with the help of tools like perf [60] and DTrace [20]. For a multi-threaded application, this task becomes much more challenging because a thread could spend time waiting for certain events (e.g., lock, I/O, condition variable, etc.) as well as executing code: both execution and waiting can create bottlenecks.

Accordingly, performance analysis tools targeting multi-threaded applications can be categorized into two types: on-CPU analysis to identify bottlenecks created by execution and off-CPU analysis to identify bottlenecks created by waiting [56]. As shown in previous works, off-CPU analysis is important because optimizing waiting can lead to a significant improvement in performance [3, 4, 9, 14, 42, 69–71, 76].

While there are systematic solutions for on-CPU analysis (e.g., Critical Path Analysis [40] and COZ [16]), existing off-CPU analysis methods are either inaccurate or incomplete. For example, a number of tools can rank waiting events based on their lengths [36, 57, 74], but longer waiting events are not necessarily more important (see Section 2); some other tools design metrics to rank lock contention [2, 18, 75], which is certainly one of the most important types of waiting events, but other waiting events, such as waiting for condition variables or I/Os, can create a bottleneck as well (also see Section 2). As far as we know, no tools can perform accurate analysis for all kinds of waiting events.

To identify waiting events critical to throughput, the key challenge is a gap between the *local impact* and the *global impact* of waiting events: given the information of a waiting event, such as its length and frequency, it may not be hard to predict its impact on the threads waiting for the event (i.e., local impact). To improve overall application throughput, however, we need to improve the throughput of all threads involved in request processing (called *worker threads* in this paper). Therefore, to understand whether optimizing a waiting event can improve overall throughput, we need to know whether its impact can reach all worker threads (i.e., global impact). These two kinds of impacts are not always correlated: events with a small local impact usually have a small global impact, but events with a large local impact may not have a large global impact. As a result, it's hard to directly rank the global impact of waiting events.

To address this problem, we propose a novel technique called "wait-for graph" to compute which threads a waiting event may influence. This technique is based on a

simple observation: if thread B never waits for thread A, either directly or indirectly, then optimizing A's waiting events would not improve B, because neither B's execution speed nor B's waiting time would be affected. Following this observation, wPerf models the application as a wait-for graph, in which each thread is a vertex and a directed edge from A to B means thread A sometimes waits for B. We can prove that if such a graph contains any *knots* with worker threads inside them, we must optimize at least one waiting event in each of these knots. Intuitively, this conclusion is a generalization of our observation: a knot is an inescapable section of the graph (see formal definition in Section 3.1), which means the worker threads in a knot never wait for outside threads, so optimizing outside events would not improve these worker threads. However, to improve overall throughput, we must improve all worker threads, which means we must optimize at least one event in the knot. In other words, each knot must contain a bottleneck.

A knot means there must exist cyclic wait-for relationship among its threads. In practice, such cyclic wait-for relationship can be caused by various reasons, such as blocking I/Os, load imbalance, and lock contention.

For complicated knots, wPerf refines them by trimming edges whose local impact is small, because events with little local impact usually have little global impact and thus optimizing them would have little impact on the application. For this purpose, the length of a waiting event can serve as a natural heuristic for its local impact, but using it directly may not be accurate when waiting events are nested. For example, if thread A wakes up B and then B wakes up C later, adding all C's waiting period to edge $C \rightarrow B$ is misleading because part of this period is caused by B waiting for A. To solve this problem, we introduce a technique called "cascaded redistribution" to quantify the local impact of waiting events: if thread A waits for thread B from t_1 to t_2, wPerf checks what B is doing during t_1 to t_2 and if B is waiting for another thread, wPerf will re-distribute the corresponding weight and perform the check recursively.

Given such local impact as a weight on each edge, wPerf can refine a complicated knot by continuously removing its edges with small weights, till the knot becomes disconnected, which allows wPerf to further identify smaller knots. wPerf repeats these two procedures (i.e., identify knots and refine knots) iteratively until the graph is simple enough, which should contain events whose local impact is large and whose impact can potentially reach all worker threads.

We apply wPerf to various open-source applications. Guided by the reports of wPerf, we are able to improve their throughput by up to 4.83×. For example, we find in ZooKeeper [34], using blocking I/Os and limiting the number of outstanding requests combined cause inefficiency when the workload is read-heavy: in this case, for each logging operation, ZooKeeper can only batch a small number of writes, leading to inefficient disk performance. wPerf's runtime overhead of recording waiting events is about 5.1% on average.

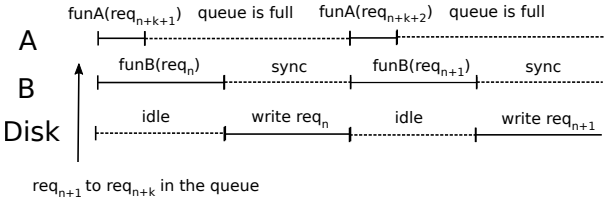
Thread A

```
while (true)
    recv req from network
    funA(req) //2ms
    queue.enqueue(req)
```

Thread B

```
while (true)
    req = queue.dequeue()
    funB(req) //5ms
    log req to a file
    sync //5ms
```

(a) Code (queue is a producer-consumer queue with max size k)



(b) Runtime execution.

Figure 1: An example of a multi-threaded program with a bottleneck waiting event.

2 Motivating Example

This section presents an example that motivates our work. As shown in Figure 1: since thread B needs to sync data to the disk (Figure 1a), B and the disk cannot process requests in parallel at runtime (Figure 1b). As a result, B and the disk combined take 10ms to process a request, which becomes the bottleneck of this application. As one can see, this application is saturated while none of its threads or disks are fully saturated. Furthermore, one can observe the following phenomena:

- Off-CPU analysis is important. In this example, on-CPU analysis like Critical Path Analysis [40] or COZ [16] can identify that *funB* and disk write are worth optimizing, which is certainly correct, but we should not neglect that the blocking pattern between B and the disk is worth optimizing as well: if we can change thread B to write to disk asynchronously, we could double the throughput of this application.
- While lock contention is well studied, we should not neglect other waiting events. The bottleneck of this example is not caused by contentions, but by waiting for I/Os. Replacing the disk with thread C and letting B wait for C on a condition variable can create a similar bottleneck.
- Longer waiting events are not necessarily more important. In other words, events with a large local impact may not have a large global impact. In this example, thread A spends 80% of its time waiting for B, which

is longer than the time B spends waiting for the disk, but it is because A has less work than B and is not the cause of the bottleneck.

Although a number of tools like off-CPU flame graph [57] have been developed to help off-CPU analysis, we are not aware of any tools that can answer the question which waiting events are important, when considering all kinds of waiting events. As a result, such investigation largely relies on the efforts of the developers. For the above simple example, it may not be difficult. In real applications, however, such patterns can become complicated, involving many more threads and devices (see Section 5). These phenomena motivate us to develop a new off-CPU analysis approach, which should be generic enough to handle all kinds of waiting events.

3 Identify Bottleneck Waiting Events

In this paper, we propose wPerf, a generic approach to identify bottleneck waiting events in multi-threaded applications. To be more specific, we assume the target application is processing requests from either remote clients or user inputs, and the goal of wPerf is to identify waiting events whose optimization can improve the application's throughput to process incoming requests.

wPerf models the target application as a number of threads (an I/O device is modeled as a pseudo thread). A thread is either executing some task or is blocked, waiting for some event from another thread. A task can be either a portion of an incoming request or an internal task generated by the application. A thread can be optimized by 1) increasing its speed to execute tasks; 2) reducing the number of tasks it needs to execute; or 3) reducing its waiting time. Since wPerf targets off-CPU analysis, it tries to identify opportunities for the third type.

To identify bottleneck waiting events, wPerf uses two steps iteratively to narrow down the search space: in the first step, it builds the wait-for graph to identify subgraphs that must contain bottlenecks. If these subgraphs are large, wPerf refines them by removing edges with little local impact.

In this section, we first present a few definitions, then explain the basic idea of wPerf in a simplified model, and finally extend the model to general applications.

3.1 Definitions

Definition 3.1. *Worker and background threads. A thread is a worker thread if its throughput of processing its tasks grows with the application's throughput to process its incoming requests; a thread is a background thread if its throughput does not grow with the throughput of the application.*



Figure 2: Wait-for graph of the application in Figure 1.

For example, threads that process incoming requests are obvious worker threads; threads that perform tasks like garbage collection or disk flushing are also worker threads, though they usually run in the background; threads that perform tasks like sending heartbeats are background threads.

This definition identifies threads that must be optimized to improve overall application throughput (i.e., worker threads), because they are directly or indirectly involved in processing incoming requests. In real applications, we find most of the threads are worker threads.

Definition 3.2. *Wait-for relationship. Thread A directly waits for thread B if A sometimes is woken up by thread B. Thread A indirectly waits for B if there exists a sequence of threads T_1, T_2, ... T_n such that $T_1 = A$, $T_n = B$, and T_i directly waits for $T_{(i+1)}$. Thread A waits for thread B if A either directly or indirectly waits for B.*

Definition 3.3. *Wait-for graph. We construct a wait-for graph for a multi-threaded application in the following way: each vertex is a thread and a directed edge from thread A to B means A directly waits for B.*

For example, Figure 2 shows the wait-for graph for the application shown in Figure 1. One can easily prove that A waits for B if there is a directed path from A to B.

Definition 3.4. *Knot and sink. In a graph, a knot is a nonempty set K of vertices such that the reachable set of each vertex in K is exactly set K; a sink is a vertex with no edges directed from it [32].*

Intuitively, knot and sink identify minimal inescapable sections of a graph. Note that by definition, a vertex with a self-loop but no other outgoing edges is a knot.

3.2 Identify bottleneck waiting events in a simplified model

In this simplified model, we make the following assumptions and we discuss how to relax these assumptions in the next section: 1) each application is running a fixed number of threads; 2) there are more CPU cores than the number of threads; 3) all threads are worker threads; 4) threads are not performing any I/O operations. Our algorithm uses two steps to narrow down the search space.

3.2.1 Step 1: Identifying knots

Our algorithm first narrows down the search space by identifying subgraphs that must contain bottlenecks, based on the following lemma and theorem.

Lemma 3.1. *If thread B never waits for A, reducing A's waiting time would not increase the throughput of B.*

Proof. If we don't optimize the execution of B, the only way to improve B's throughput is to give it more tasks, i.e., reduce its waiting time. However, since B never waits for A, optimizing A would not affect B's waiting time. Therefore, B's throughput is not affected. □

Theorem 3.2. *If the wait-for graph contains any knots, to improve the application's throughput, we must optimize at least one waiting event in each knot.*

Proof. We prove by contradiction: suppose we can improve the application's throughput without optimizing any events in a knot. On one hand, since all threads are worker threads, if overall throughput were improved, the throughput of each thread should increase (Definition 3.1). On the other hand, because a knot is an inescapable section of a graph, threads in the knot never wait for outside threads, so optimizing outside threads or events would not improve the throughput of threads in the knot (Lemma 3.1). These two conclusions contradict and thus the theorem is proved. □

For example, in Figure 2, thread B and the disk form a knot and thus at least one of their waiting events must be optimized to improve the application's throughput.

A graph must contain either knots or sinks or both [32]. A sink means the execution of the corresponding thread is the bottleneck, which is beyond the scope of off-CPU analysis. A knot means there must exist cyclic wait-for relationship among multiple threads, which can cause the application to saturate while none of the threads on the cycle are saturated. In practice, such cyclic wait-for relationship can happen for different reasons, among which the following ones are common:

- Lock contention. Multiple threads contending on a lock is probably the most common reason to cause a cyclic wait-for relationship. In this case, threads contending on the lock may wait for each other.
- Blocking operation. Figure 1 shows an example of this problem: since B needs to wait for the responses from the disk, and the disk needs to wait for new requests from B, there exists a cyclic wait-for relationship between B and the disk.
- Load imbalance. Many applications work in phases and parallelize the job in each phase [19, 68]. Imbalance across phases or imbalance across threads in the same phase can create a cycle. For example, suppose in phase 1, thread A executes three tasks and thread B executes one task; in phase 2, A executes one task and B executes three tasks: in this case, A needs to wait for B at the end of phase 1 and B needs to wait for A at the end of phase 2, creating a cycle.

3.2.2 Step 2: Refining knots

If a knot is small, the developers may manually investigate it and decide how to optimize. For a large knot, wPerf further narrows down the search space by removing edges whose optimization would have little impact on the application. However, accurately predicting the global impact of a waiting event is a challenging problem in the first place. To address this challenge, we observe that the local impact of a waiting event can be viewed as the *upper bound* of the global impact of this event: improvement to all threads naturally includes improvement to threads waiting for this event, so the local impact of an event should be at least as large as its global impact.

Following this observation, wPerf removes edges with a small local impact until the knot becomes disconnected. When disconnection happens, wPerf tries to identify smaller knots. wPerf repeats these two procedures—identifying knots and trimming edges with a small local impact—until the result is simple enough for developers. We discuss the termination condition in Section 4.3. By combining these two procedures, wPerf essentially tries to identify the edges with a large impact on all worker threads.

Since local impact marks the upper bound of global impact, knot refinement will not bring false negatives (i.e., removing important edges), which means the user will not miss important optimization opportunities. However, it may bring false positives (i.e., not removing unimportant edges), which requires additional effort from the user, but in our case studies, we find such additional effort is not significant, mainly because many edges with a large local impact are outside of the knot and thus are removed.

The total waiting time spent on an edge is a natural heuristic to quantify the local impact of the edge, but we find it may be misleading when waiting events are nested. To illustrate the problem, we show an example in Figure 3: thread C wakes up B at time t_1 and B wakes up A at time t_2. In practice, such nested waiting can happen in two ways: first, it is possible that C wakes up B and A simultaneously and B happens to execute first (e.g., C releases a lock that both A and B try to grab) and we call this type "symmetric waiting"; second, it is also possible that A's Task 3 depends on B's Task 2, which depends on C's Task 1. We call this type "asymmetric waiting". However, from the recorded waiting events, wPerf does not know which type it is, which means its solution to compute the edge weights should work for both types.

To motivate wPerf's solution, we show several options we have tried. The naive solution (Graph1) adds weight $(t_2 - t_0)$ to edge $A \rightarrow B$ and weight $(t_1 - t_0)$ to edge $B \rightarrow C$. This solution underestimates the importance of $B \rightarrow C$, because reducing the time spent on

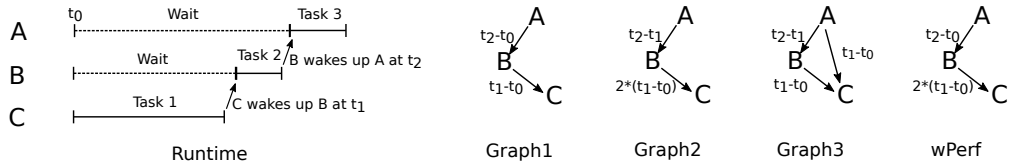


Figure 3: Building edges weights from length of waiting events (Graphs 1-3 are our failed attempts).

$B \rightarrow C$ can automatically reduce the time spent on $A \rightarrow B$. Graph2 moves the overlapping part $(t_1 - t_0)$ from $A \rightarrow B$ to $B \rightarrow C$, which increases the importance of $B \rightarrow C$, but it underestimates the importance of $A \rightarrow B$: in asymmetric waiting, it is possible to optimize $A \rightarrow B$ but not optimize $B \rightarrow C$, so it is inappropriate to assume optimizing $A \rightarrow B$ can only reduce the waiting time by $t_2 - t_1$. Graph3 draws a new edge $A \rightarrow C$ and moves the weight of $(t_1 - t_0)$ to the new edge, indicating that $(t_1 - t_0)$ is actually caused by waiting for C: this approach makes sense for symmetric waiting, but is confusing for asymmetric waiting, in which A does not directly wait for C. wPerf's solution is to keep weight $(t_2 - t_0)$ for edge $A \rightarrow B$, which means optimizing this edge can reduce A's waiting time by up to $(t_2 - t_0)$, and increases the weight of $B \rightarrow C$ by $(t_1 - t_0)$, which means optimizing this edge can lead to improvement in both B and A. wPerf's solution may seem to be unfair for symmetric waiting, but for symmetric waiting, A and B should have similar chance to be woken up first, so if we test the application for sufficiently long, the weights of $A \rightarrow B$ and $B \rightarrow C$ should be close.

Following this idea, wPerf introduces a cascaded redistribution algorithm to build the weights in the general case: at first, wPerf assigns a weight to an edge according to the waiting time spent on that edge. If wPerf finds while thread A is waiting for thread B, thread B also waits for thread C (length t), wPerf increases the weight of $(B \rightarrow C)$ by t. If C waits for other threads during the same period of time, wPerf will perform such adjustment recursively (see the detailed algorithm in Section 4.2).

3.3 Extending the model

Next, we extend our model by relaxing its assumptions.

Not enough CPUs. A thread may also wait because all CPU cores are busy (i.e., the thread is in "runnable" state). We can record the runnable time of each thread: if a thread in a knot is often in the runnable state, then the application may benefit from using more CPU cores or giving those bottleneck threads a higher priority.

I/Os. wPerf models an I/O device as a pseudo thread. If a normal thread sometimes waits for an I/O to complete, wPerf draws an edge from the normal thread to the corresponding I/O thread. If an I/O device is not fully utilized (see Section 4.1), wPerf draws an edge from the I/O thread to all normal threads that have issued I/Os to this device, meaning the device waits for new I/Os from these normal threads.

Busy waiting. Some threads use busy waiting to continuously check whether another thread has generated the events. A typical example is a spin lock. From the OS point of view, a thread that is busy waiting is not counted as waiting, because it is executing code; at the logical level, however, time spent on busy waiting should be counted as waiting time in our model. We discuss how to trace such events in Section 4.1.

Background threads. A knot consisting of only background threads does not have to be optimized to improve the application's throughput, because the throughput of a background thread does not grow with the application's throughput. Note that though not necessary, optimizing such a knot may still be beneficial. For example, suppose a background thread needs to periodically send a heartbeat, during which it needs to grab a lock and thus may block a worker thread. In this case, reducing the locking time of the background thread may improve the worker threads contending on the same lock, but it is not ncessary since optimizing those worker threads may improve the application's throughput as well. Therefore, wPerf reports such a knot to the user, removes the knot, and continues to analyze the remaining graph, because there may exist other optimization opportunities. wPerf uses the following heuristic to identify such a knot: if the knot does not contain any I/O threads and the sum of the CPU utilization of all threads in the knot is less than 100%, wPerf will report it, because this means some threads in the knot sleep frequently, which is a typical behavior of background threads.

Short-term threads. Some applications create a new thread for a new task and terminate the thread when the task finishes. Such short-term threads do not follow our definition of worker thread, because their throughput does not grow with the application's throughput. To apply our idea, wPerf merges such short-term threads into a virtual long-running thread: if any of the short-term threads is running/runnable, wPerf marks the virtual thread as running/runnable; otherwise, wPerf marks the virtual thread as blocked, indicating it is waiting for new tasks from the thread that is creating these short-term threads.

4 Design and Implementation

To apply the above ideas, wPerf incorporates three components: the recorder records the target application's and the OS's waiting events at runtime; the controller receives commands from the user and sends the commands to the recorder; the analyzer builds the wait-for graph from the recorded events offline and tries to identify knots or sinks. In this section, we present how recorder and analyzer work in detail.

4.1 Recording sufficient information

The responsibility of the recorder is to capture sufficient information to allow the analyzer to build the wait-for graph. Towards this goal, such information should be able to answer two questions: 1) if a thread is waiting, which thread is it waiting for? and 2) how long does a thread spend on waiting for another thread? The former will allow us to create edges in the wait-for graph, and the latter will allow us to compute weights for edges.

Profiling tools (e.g., perf [60], DTrace [20], ETW [1], etc.) can record events at different layers. We decide to record waiting events at low layers (i.e. CPU scheduling and interrupt handling) because events at lower layers usually can provide more accurate answers to the above two questions. Taking I/O waiting as an example, one option is to record the lengths of related system calls, but such information is not precise: it is possible that most of the time is indeed spent on waiting for I/Os to complete; it is possible that much time is spent on in-kernel processing, such as data copy; it is also possible that in the kernel, this system call contends with another thread (e.g., write to the same file). Recording at lower layers, on the other hand, can provide precise information.

Following this observation, wPerf uses *kprobe* [41] to record key waiting events in the kernel, with one exception about busy waiting. Since we implement wPerf on Linux, next we first present the background about how Linux performs scheduling and interrupt handling and then present what information wPerf records.

Background. A thread can be in different states: a thread is *running* if it is being executed on a CPU; a thread is *runnable* if it is ready to run but has not been scheduled yet, maybe because all CPUs are busy; a thread is *blocked* if it is waiting for some events and thus cannot be scheduled. While an application can block or unblock a thread through corresponding system calls, OS scheduling module decides which threads to run.

When an interrupt is triggered, CPU jumps to the predefined interrupt request (IRQ) function, preempting the current thread running on the CPU. An IRQ function is usually not executed in a thread context, so it is not controlled by scheduling, which means wPerf has to record IRQ events as well as scheduling events. An IRQ function can wake up a blocked thread: this is common when the thread is waiting for I/Os to complete.

Recording scheduling events. For CPU scheduling, wPerf records two key functions: *__switch_to* and *try_to_wake_up*. *try_to_wake_up* changes a thread's state from *blocked* to *runnable*, which can be invoked in functions like *pthread_mutex_unlock* or when an I/O completes (usually in an IRQ). For this function, wPerf records the timestamp, the thread ID of the thread to be woken up, and the entity (either a thread or an IRQ) that invokes the wakeup. *__switch_to* switches out a thread from a CPU and switches in another. The thread that is switched in must be in *running* state; the one that gets switched out could be either in *runnable* state, which means this switch is caused by CPU scheduling, or in *blocked* state, which means this switch is caused by events like *pthread_mutex_lock* or issuing an I/O. wPerf records the timestamp and the states of both threads.

Recording IRQ events. wPerf intercepts IRQ functions to record its starting time, ending time, its type, and which CPU it runs. To know IRQ type, wPerf intercepts soft IRQ functions defined in *interrupt.h*, each for a specific type of device. By utilizing the function name, wPerf can know what type of hardware device triggers the interrupt, but this approach has a limitation that it cannot distinguish different instances of the same type of devices. This problem could be solved if wPerf can record the IRQ number, which is unique to each device, but unfortunately in Linux, IRQ number is not observable to every IRQ function. Modifying Linux kernel could solve this problem, but our current implementation tries to avoid kernel modification for portability.

Recording information for I/O devices. wPerf models an I/O device as a pseudo I/O thread (Section 3.3). To build the wait-for graph, wPerf needs to know 1) how long a normal thread waits for an I/O thread and 2) how long an I/O thread waits for a normal thread. The recorded IRQ events can only answer the first question.

Since we cannot instrument the internal execution of a hardware device, we have designed an approximate solution to answer the second question: we assume an I/O device is waiting during its idle time; we draw an edge from the device to each normal thread that has issued an I/O to this device; and we distribute the device's idle time to different edges based on how much data each thread sends to the device, meaning the device is waiting for new I/Os from these threads in its idle time. To implement this mechanism, we need to estimate the idle time of each device.

For a disk, we record its used bandwidth and I/Os per second (IOPS). We use the bandwidth to estimate the disk's idle time under sequential I/Os and use the

IOPS to estimate its idle time under random I/Os. The case about network interface card (NIC) is more complicated because its capacity is not only limited by the NIC device, but also by the network infrastructure or the remote service. Our current implementation uses the NIC's maximal bandwidth as an upper bound to estimate the NIC's idle time. If the user has a better knowledge about the link bandwidth or the capacity of the remote service, wPerf can use these values for a better estimation.

Recording information for busy waiting. From the OS point of view, a thread that is performing busy waiting is in *running* state but logically it is in *blocked* state. Since such waiting and waking up do not involve kernel functions, recording events in kernel cannot capture them. To make things worse, there is no well-defined interface for such mechanism: some applications use spinlock provided by *pthread* while others may implement their own mechanisms (e.g., MySQL [51]). Previous studies have shown that, although such mechanisms are error prone, they are quite popular [73].

wPerf has no perfect solution to this problem. Instead, it relies on the developers' knowledge. wPerf provides two tracing functions *before_spin* and *after_spin* to developers, so that they can insert these tracing functions at appropriate places. In practice, a developer does not need to trace every of such functions. Instead, he/she can first find frequent ones with on-CPU analysis tools, and then instrument these frequent ones.

Removing false wakeup. A false wakeup is a phenomenon that a thread is woken up but finds its condition to continue is not satisfied, so it has to sleep again. For example, a ticket selling thread A may broadcast to threads B and C, claiming it has one ticket. In this case, only one of B and C can get the ticket and continue. Suppose B gets the ticket: though wPerf can record an event A waking up C, adding weight to edge $C \rightarrow A$ is misleading, because C's condition to continue is not satisfied.

Similar as the case for busy waiting, wPerf provides a tracing function to developers, which can declare a wakeup event as a false one. The developer can insert it after a wakeup, together with a condition check. During analysis, wPerf will remove the pair of wakeup and waiting events that encapsulate this declaration. Once again, the developer only needs to identify significant ones.

Recording call stacks. Developers need to tie events to source code to understand the causes of waiting. For this purpose, wPerf utilizes perf [60] to sample the call stacks of the scheduling and IRQ events as mentioned above. By comparing the timestamp of a call stack with the timestamps of recorded events, wPerf can affiliate a call stack to an edge in the wait-for graph to help developers understand why each edge occurs. Note that getting accurate call stacks requires additional supports, such as enabling the sched_schedstats feature in kernel and compiling C/C++ applications with the -g option. For Java applications, we need to add the -XX:+PreserveFramePointer option to the JVM and attach additional modules like perf-map-agent [61] or async-profiler [6] (wPerf uses perf-map-agent). We are not aware of supports for Python applications yet.

Minimizing recording overhead. To reduce recording overhead, we apply two classic optimizations: 1) to reduce I/O overhead, the recorder buffers events and flushes the buffers to trace files in the background; 2) to avoid contentions, the recorder creates a buffer and a trace file for each core. Besides, we meet two challenges.

First, recording busy waiting and false wakeup events can incur a high overhead in a naive implementation. The reason is that these events are recorded in the user space, which means a naive implementation needs to make system calls to read the timestamp and the thread ID of an event: frequent system calls are known to have a high overhead [65]. To avoid reading timestamps from the kernel space, we use the virtual dynamic shared object (vDSO) technique provided by Linux to read current time in the user space; to avoid reading thread ID from the kernel space, we observe the *pthread* library provides a unique *pthread* ID (PID) for each thread, which can be retrieved in the user space. However, recording only PIDs is problematic, because PID is different from the thread ID (TID) used in the kernel space. To create a match between such two types of IDs, the recorder records both PID and TID for the first user-space event from each thread and records only PIDs afterwards.

Second, Linux provides different types of clocks, but the types supported by vDSO and perf have no overlap, so we cannot use a single type of clock for all events. To address this problem, the recorder records two clock values for each kernel event, one from the vDSO clock and one from the perf clock. This approach allows us to tie perf call stacks to kernel events and to order user-space events and kernel events. However, this approach cannot create an accurate match between perf call stacks and user-space events, so we decide not to record call stacks for user-space events: this is fine since the user needs to annotate these events anyway, which means he/she already knows the source code tied to such events.

4.2 Building the wait-for graph

Based on the information recorded by the recorder, wPerf's analyzer builds the wait-for graph and computes the weights of edges offline in two steps.

In the first step, the analyzer tries to match *wait* and *wakeup* events. A *wait* event is one that changes a thread's state from "running" or "runnable" to

```
1  input: w is a waiting segment.
2  w.start: starting time of this segment
3  w.end: ending time of this segment
4  w.ID: thread ID of this segment
5  w.wakerID: the thread that wakes up this
       segment

7  function cascade(w)
8      add weight (w.end-w.start) to edge
           w.ID → w.wakerID
9      find all waiting segments in w.wakerID
           that overlap with [w.start w.end)
10     for each of these segments
11         if segment.start < w.start
12             segment.start = w.start
13         if segment.end > w.end
14             segment.end = w.end
15         cascade(segment)
```

Figure 4: Pseudocode of cascaded re-distribution.

"blocked"; a *wakeup* event is one that changes a thread's state from "blocked" to "runnable". For each *wait* event, the analyzer searches for the next *wakeup* event that has the waiting thread's ID as the argument.

Such matching of *wait* and *wakeup* events can naturally break a thread's time into multiple segments, in either "running/runnable" or "waiting" state. The analyzer treats running and runnable segments in the same way in this step and separates them later. At the end of this step, the analyzer removes all segments which contain the false wakeup event, by removing the *wakeup* and *wait* events that encapsulate the event.

In the next step, the analyzer builds the wait-for graph using the cascaded re-distribution algorithm (Figure 3). As shown in Figure 4, the analyzer performs a recursive algorithm for each waiting segment: it first adds the length of this segment to the weight of edge $w.ID \rightarrow w.wakerID$ (line 8) and then checks whether thread wakerID is waiting during the same period of time (line 9). If so, the analyzer recursively calls the *cascade* function for those waiting segments (line 15). Note that the waiting segments in wakerID will be analyzed as well, so their lengths are counted multiple times in the weights of the corresponding edges. This is what cascaded re-distribution tries to achieve: nested waiting segments that cause multiple threads to wait should be emphasized, because optimizing such segments can automatically reduce waiting time of multiple threads.

After building the wait-for graph, the analyzer applies the algorithms described in Section 3: the analyzer first applies the Strongly Connected Component (SCC) algorithm to divide the graph into multiple SCCs and finds SCCs with no outgoing edges: an SCC with no outgoing edges is either a knot or a sink. If a knot is still complex, the analyzer repeatedly removes the edge with the lowest weight, until the knot becomes disconnected. Then the analyzer identifies knots or sinks again. The analyzer repeats this procedure till the developer finds the knot understandable. Finally, the analyzer checks whether the remaining threads contain any runnable segments: if so, the application may benefit from using more CPU cores or giving higher priority to these threads.

The analyzer incorporates two optimizations:

Parallel graph building. Building the wait-for graph could be time consuming if the recorded information contains many events. The analyzer parallelizes the computation of both steps mentioned above. In the first step, the analyzer parallelizes the matching of events and separation of segments: this step does not require synchronization because the event list is read-only and the output segment information is local to each analyzer thread. In the second step, the analyzer parallelizes the cascaded re-distribution for each segment: this phase does not require synchronization either because the segmentation information becomes read-only and we can maintain a local wait-for graph for each analyzer thread and merge all local graphs when all threads finish.

Merging similar threads. Many applications create a number of threads to execute similar kinds of tasks. wPerf merges such threads into a single vertex to simplify the graph. To identify similar threads, wPerf's utilizes the recorded call stacks: the analyzer merges two threads if their distributions of call stacks are similar. Note that in the original wait-for graph, a vertex should never have a self-loop because a thread should not wait for itself, but after merging similar threads, a self-loop can happen if similar threads wait for each other.

4.3 Using wPerf

First, the user needs to run the target application and use the wPerf recorder to record events. wPerf provides commands to start and stop recording at any time. If the user observes significant busy waiting or false wakeup during the experiment, he/she should annotate those events and re-run the experiment.

Then the user needs to run the analyzer on the recorded events. The analyzer provides both a graphic output and a text output to present the bottleneck. In this step, the user can set up the termination condition of knot refinement. By default, the refinement terminates when the remaining graph is either a single vertex or a simple cycle. In addition, the user can instruct the refinement to terminate when the smallest weight in the remaining graph is larger than a threshold. The user should set this threshold based on how much improvement he/she targets, since the weight of an edge represents the upper bound of the improvement one may gain by optimizing the edge.

In the third step, the user needs to investigate the knot to identify optimization opportunities. To facilitate such investigation, wPerf allows the user to query the call

	Problem	Speedup	Known fixes?	Involved techniques
HBase [5]	Blocking write	2.74×	Yes	VI, M-SHORT, M-SIM, FW
ZooKeeper [34, 79]	Blocking write	4.83×	No	VI
HDFS [29, 64]	Blocking write	2.56×	Yes	VI, M-SIM
NFS [55]	Blocking read	3.9×	No	VI, M-SIM
BlockGrace [10, 72]	Load imbalance	1.44×	No	M-SHORT, M-SIM
Memcached [47]	Lock contention	1.64×	Partially	VI, M-SIM
MySQL [51]	Lock contention	1.42×	Yes	VI, M-SIM, BW

Table 1: Summary of case studies. (Speedup = $\frac{ImprovedThroughput}{OriginalThroughput}$; VI: virtual I/O threads; M-SHORT: merging short-term threads; M-SIM: merging similar threads; BW: tracing busy waiting; FW: tracing false wakeup)

stacks attached to each edge to understand how each edge is formed. This step requires the user's efforts, and our experience is that for one who is familiar with the target application, this step usually takes no more than a few hours. One reason that simplifies this step is that many edges are caused by a thread waiting for new tasks from another thread (e.g., $Disk \rightarrow B$ in Figure 1), which are usually not optimizable.

Finally, the user needs to optimize the application. Similar as most other profiling tools, wPerf does not provide any help in this step. Based on our experience (Section 5), we have summarized a few common problems and potential solutions, most of which are classic: for blocking I/Os, one could consider using non-blocking I/Os or batching I/Os; for load imbalance, one could consider fine-grained task scheduling; for lock contention, one could consider fine-grained locking. However, since most of such optimizations will affect the correctness of the application, the user needs to investigate whether it is possible and how to apply them. In our case studies, the required user's efforts in this step vary significantly depending on the optimization, ranging from a few minutes to change a configuration option to a few weeks to re-design the application.

Taking the application in Figure 1 as an example, wPerf will output a wait-for graph like Figure 2, in which B and the disk form a knot. The user can then query the call stacks of edges $B \rightarrow Disk$ and $Disk \rightarrow B$; wPerf will show that $B \rightarrow disk$ is caused by the *sync* call in thread B and $Disk \rightarrow B$ is caused by the disk waiting for new I/Os from B. The user will realize that $Disk \rightarrow B$ is not optimizable and thus will focus on the *sync* call.

5 Case Study

To verify the effectiveness of wPerf, we apply wPerf to a number of open-source applications (Section 5.1): we try to optimize the events reported by wPerf and see whether such optimization can lead to improvement in throughput. We find some problems are already fixed in newer versions of the applications or online discussions, which can serve as a direct evidence of wPerf's accuracy. Table 1 summarizes our case studies. Note that we have avoided complicated optimizations because how to optimize is not the contribution of wPerf, and thus there may exist better ways to optimize the reported problems.

Furthermore, as a comparison, we run three existing tools on the same set of applications and present their reports (Section 5.2). Finally, we report the overhead of online recording and offline analysis (Section 5.3).

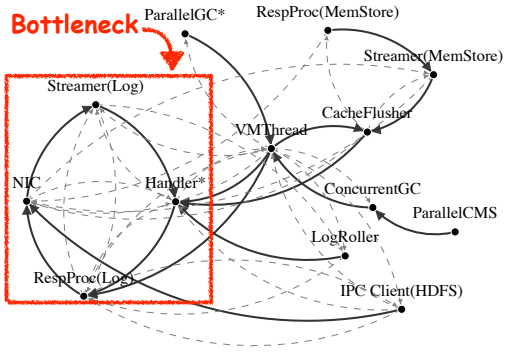
We run all experiments in a cluster with 21 machines: one machine is equipped with two Intel Xeon E5-2630 8-core processors (2.4GHz), 64GB of memory, and a 10Gb NIC; 20 machines are equipped with an Intel Xeon E3-1231 4-core processor (3.4GHz), 16GB of memory, and a 1Gb NIC each.

For each experiment, we record events for 90 seconds. We set the analyzer to terminate when the result graph is a single vertex or a simple cycle or when the lowest weight of its edges is larger than 20% of the recording time (i.e., 18). We visualize the wait-for graph with D3.js [17], and we use solid lines to draw edges whose weights are larger than 18 and use dashed lines to draw the other edges. Since D3.js cannot show a self-loop well, we use "*" to annotate threads with self-loops whose weights are larger than 18. We record all edge weights in the technical report [78]. wPerf uses a thread ID to represent each thread, and for readability, we manually check the call stacks of each thread to find its thread name and replace the thread ID with the thread name. We set perf sampling frequency to be 100Hz, which allows perf to collect sufficient samples with a small overhead.

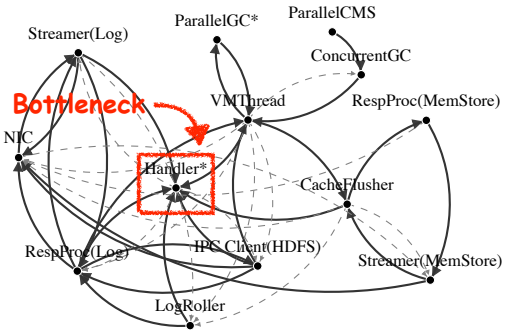
5.1 Effectiveness of wPerf

HBase. HBase [5] is an open-source implementation of Bigtable [12]. It provides a key-value like interface to users and stores data on HDFS. We first test HBase 0.92 with one RegionServer, which runs on HDFS with three DataNodes. We run a write workload with a key size of 16 bytes and a value size of 1024 bytes.

With the default setting, HBase can achieve a through-



(a) HBase with 10 handlers (default setting)

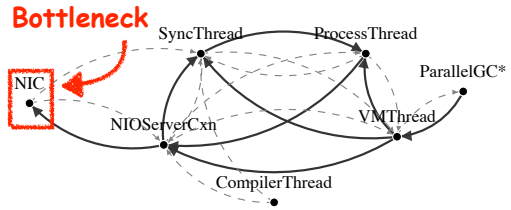


(b) HBase with 60 handlers

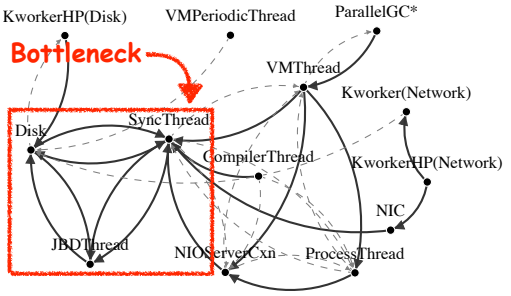
Figure 5: Wait-for graphs of HBase. For readability, we sort edges by their weights and only show the top 40.

put of 9,564 requests per second (RPS). Figure 5a shows the wait-for graph, in which wPerf identifies a significant cycle among HBase Handler threads, HDFS Streamer threads, the NIC, and HDFS ResponseProcessor threads. This cycle is created for the following reason: the Handler threads flushes data to the Streamer threads; the Streamer threads send data to DataNodes through the NIC; when the NIC receives the acknowledgements from the DataNodes, it wakes up the ResponseProcessors; and finally the ResponseProcessors notify the Handlers that a flushing is complete. The blocking flushing pattern, i.e., the Handlers must wait for notification of flushing complete from the ResponseProcessor, is the fundamental reason to create the cycle. The HBase developers are aware that blocking flush is inefficient, so they create multiple Handlers to flush in parallel, but the default number of 10 Handlers is too small on a modern server.

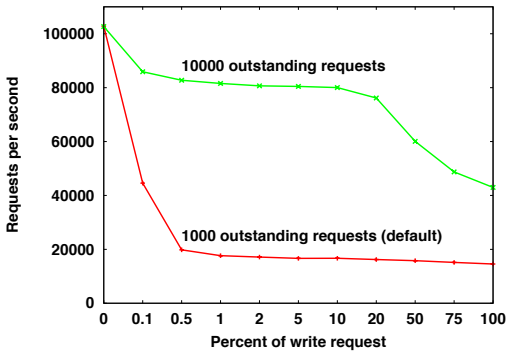
We increase the number of Handlers and HBase can achieve a maximal throughput of 13,568 RPS with 60 Handlers. Figure 5b shows the new wait-for graph, in which wPerf identifies the Handlers as the main bottleneck. Comparing to Figure 5a, the edge weight of Handler → ResponseProcessor drops from 87.4 to 16.5: this is because overlapping more Handlers make them spend more time in runnable state. The setting of Handler count has been discussed online [27, 28].



(a) ZooKeeper with read-only workload



(b) ZooKeeper with 0.1% write workload



(c) Throughput of ZooKeeper.

Figure 6: Wait-for graphs and throughput of ZooKeeper.

In Figure 5b, wPerf identifies a significant self-loop inside Handlers. Such waiting is caused by contentions among Handlers. We find that HBase 1.28 has incorporated optimizations to reduce such contentions and our experiments show that it can improve the throughput to 26,164 RPS. Such results confirm the report of wPerf: fixing the two bottlenecks reported by wPerf can bring a total of 2.74× speedup.

ZooKeeper. ZooKeeper [34, 79] is an open-source implementation of Chubby [11]. We evaluate ZooKeeper 3.4.11 with a mixed read-write workload and 1KB key-value pairs. As shown in Figure 6c, we find a performance problem that even adding 0.1% write can significantly degrade system throughput from 102K RPS to about 44K RPS. We use wPerf to debug this problem.

As shown in Figure 6a, for the read-only workload, wPerf identifies NIC as the major bottleneck, which is reasonable because the NIC's max bandwidth is 1Gbps: this is almost equal to 102K RPS. For the workload with 0.1% write (Figure 6b), however, wPerf identifies the

key bottleneck is a knot consisting of the SyncThread in ZooKeeper, the disk, and the journaling thread in the file system. As shown in the knot, the disk spends a lot of time waiting for the other two, which means the disk's bandwidth is highly under-utilized.

We investigate the code of SyncThread. SyncThread needs to log write requests to disk and perform a blocking *sync* operation, which explains why it needs to wait for the disk. Sync for every write request is obviously inefficient, so ZooKeeper performs a classic batching optimization that if there are multiple outstanding requests, it will perform one sync operation for all of them. In ZooKeeper, the number of requests to batch is limited by two parameters: one is a configuration option to limit the total number of outstanding requests in the server (default value 1,000), which is used to prevent out of memory problems; the other is a hard-coded 1,000 limit, which means the SyncThread will not batch more than 1,000 requests. However, we find both limits count both read and write requests, so if the workload is dominated by reads, the SyncThread will only batch a small number of writes for each sync, leading to inefficient disk access.

We try a temporary fix to raise this limit to 10,000, by modifying both the configuration file and the source code. As shown in Figure 6c, such optimization can improve ZooKeeper's throughput by up to 4.83X. However, a fixed limit may not be a good solution in general: if the workload contains big requests, a high limit may cause out of memory problems; if the workload contains small requests, a low limit is bad for throughput. Therefore, it may be better to limit the total size of outstanding requests instead of limiting the total number of them.

HDFS NameNode. HDFS [29, 64] is an open-source implementation of Google File System [23]. It incorporates many DataNodes to store file data and a NameNode to store system metadata. Since NameNode is well-known to be a scalability bottleneck [63], we test it with a synthetic workload [62]: we run MapReduce TeraSort over HDFS 2.7.3, collect and analyze the RPC traces to NameNode, and synthesize traces to a larger scale.

With the default setting, NameNode can reach a maximal throughput of 3,129 RPCs per second. As shown in Figure 7, wPerf identifies the bottleneck is a cycle between Handler threads and the disk. Our investigation shows that its problem is similar to that of ZooKeeper: Handler threads need to log requests to the disk and to improve performance, NameNode batches requests from all Handlers. Therefore, the number of requests to be batched is limited by the number of Handlers. The default setting of 10 Handlers is too small to achieve good disk performance. By increasing the number of Handlers, NameNode can achieve a throughput of about 8,029 RPCs per second with 60 handlers. This problem has been discussed online [52, 53].

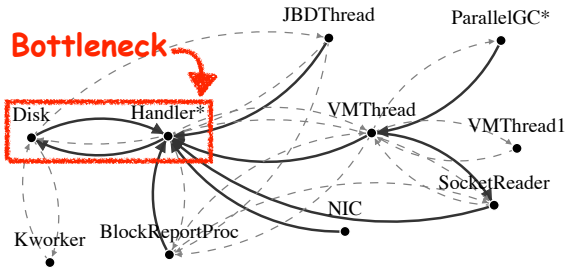


Figure 7: Wait-for graphs of HDFS NameNode.

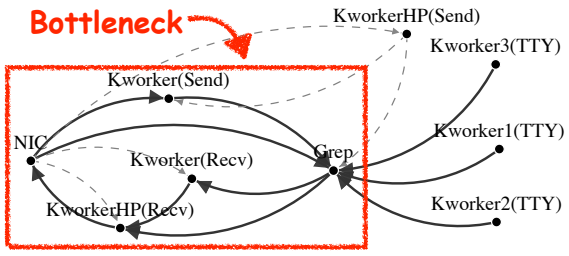


Figure 8: Wait-for graph of running grep over NFS.

NFS. Networked File System (NFS) [55] is a tool to share files among different clients. We set up an NFS server 3.2.29 on CloudLab [15] and set up one NFS client 2.7.5 on our cluster. We test its performance by storing Linux 4.1.6 kernel source code on it and running "grep".

As shown in Figure 8, wPerf identifies a cycle among the grep process, the kernel worker threads, and the NIC. The reason is that grep performs blocking read operations. As a result, grep needs to wait for data from the receiver threads, and the sender threads need to wait for new read requests from grep. This problem can be optimized by either performing reads in parallel or prefetching data asynchronously. We create two NFS instances, distribute files into them, and run eight grep processes in parallel: this can improve the throughput by 3.9×.

BlockGrace. BlockGrace [10, 72] is an in-memory graph processing system. It follows the classic Bulk Synchronous Parallel (BSP) model [68], in which an algorithm is executed in multiple iterations: in each iteration, the algorithm applies the updates from the last iteration and generates updates for the next iteration. We test BlockGrace with its own Single-Source Shortest Path (SSSP) benchmark and with 32 worker threads.

wPerf identifies a cycle between the main thread and the computation threads. Since the wait-for graph is simple, consisting of only these two types of threads, we do not show it here. Our investigation shows the primary reason is the main thread needs to perform initialization work for the computation threads, so the computation threads need to wait for initialization to finish and the main thread then waits for all computation threads to finish. To solve this problem, we let the computa-

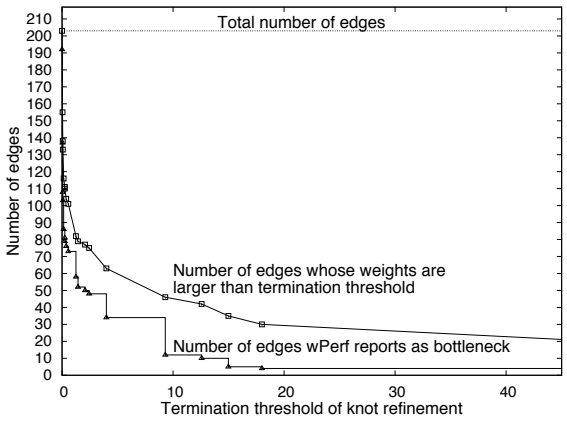


Figure 9: Changing the termination condition for HBase.

tion threads perform initialization in parallel: this can improve the throughput by 34.19%.

Then we run wPerf again and find the cycle still exists, but the weight of (computation thread → main thread) is reduced. Our investigation shows the secondary reason is the load imbalance among computation threads. To alleviate this problem, we apply fine-grained task scheduling and implement long running computation threads (original BlockGrace creates new computation threads in each iteration): these techniques can further improve the throughput by 17.82% (44.14% in total).

Memcached and MySQL. wPerf identifies contentions in these two applications. Since contention is well explored, we briefly describe our experience here and one can refer to the technical report [78] for details.

When running the memaslap benchmark [46] on Memcached 1.4.36, wPerf identifies a knot consisting of worker threads, which is mainly caused by lock contention on the LRU list and on the slab memory allocator. Optimizing them with fine-grained locking can improve the throughput from 354K RPS to 547K RPS. Memcached 1.5.2 has reduced LRU-related contention and can reach a throughput of 527K RPS; optimizing the slab allocator can improve its throughput to 580K RPS.

When running the TPC-C benchmark [67] over MySQL 5.7.20 [51] on RAM-disk, wPerf identifies a knot consisting of the worker threads, which is caused by contentions among worker threads. These contentions are caused by multiple reasons, and we are able to reduce the contention on the buffer pages by allocating more pages: this can improve the throughput from 2,682 transactions per second (TPS) to 3,806 TPS. Another major reason is contention on rows, and since previous works have studied this problem [70, 71], we do not continue.

Effects of termination condtion. The user can terminate the knot refinement when the minimal weight of the edges in the knot is larger a threshold. We use threshold 18 in previous experiments and Figure 9 uses HBase as an example to study how this threshold affects wPerf. The gap between the top line (i.e., total number of edges) and the middle line (i.e., number of edges whose weights are larger than the termination threshold) represents the number of edges eliminated because of their small weights. As one can see, only using weights as a heuristic can eliminate many edges, but even with a large threshold, there are still 20-30 edges remaining. The gap between the middle line and the bottom line (i.e., number of edges wPerf reports as bottleneck) represents the number of edges eliminated by knot identification, i.e, these edges have large weights but are outside of the knot. By combining weights (i.e., cascaded re-distribution) and knot identification, wPerf can narrow down the search space to a small number of edges. For other applications, we observe the similar trend in ZooKeeper, HDFS NameNode, NFS, and MySQL experiments; for BlockGrace and Memcached experiments, we do not observe such trend because their wait-for graphs are simple and need little refinement.

	COZ	Flame graph	SyncPerf
HBase	-	Yes	-
ZooKeeper	-	No	-
HDFS	-	No	-
NFS	No	Yes	No
BlockGrace-1	Yes	Yes	No
BlockGrace-2	No	Yes	No
Memcached	Maybe	No	Yes
MySQL	Maybe	No	*

Table 2: Can other tools identify similar problems? (- the tool does not support Java; * experiment reports errors.)

Summary. By utilizing wPerf, we are able to identify bottleneck waiting events in a variety of applications and improve their throughput, which confirms the effectiveness of wPerf. Though most of the problems we find are classic ones, they raise some new questions: many problems are caused by inappropriate setting (e.g., number of threads, number of outstanding requests, task granularity, etc.) and no fixed setting can work well for all workloads, so instead of expecting the users to find the best setting, it may be better for the application to change such setting adaptively according to the workload.

5.2 Comparison to existing tools

As a comparison, we test one on-CPU analysis tool (COZ) and two off-CPU analysis tools (perf and SyncPerf) on the same set of applications. Since COZ and SyncPerf currently do not support Java, we run them only on NFS, BlockGrace, Memcached, and MySQL. We summarize their reports in Table 2 and record all their

detailed reports in the technical report [78].

COZ. To compute how much improvement we can gain by optimizing a certain piece of code, COZ [16] *virtually speeds up* the target piece of code by keeping its speed unchanged and slowing down other code when the target code is running. After the experiment is finished, COZ adjusts the measured throughput to compensate for this slowdown.

COZ is designed for on-CPU analysis, and when we try to use it to analyze off-CPU events, we meet two problems: first, COZ's implementation can only virtually speed up execution on the CPU but cannot virtually speed up I/O devices and thus it does not report any bottlenecks related to I/Os. For example, in the grep over NFS experiment, COZ suggests us to optimize code in *kwset.c*, which is grep's internal data structure, but does not report anything related to I/Os. However, we believe there is nothing fundamental to prevent COZ from implementing virtual speed up for I/O devices. The second problem, however, is fundamental: the virtual speed up idea does not work well with waiting events, because in many cases, slowing down other events will automatically slow down a waiting event, which breaks COZ's idea to keep the speed of the target event unchanged. Taking the application in Figure 1 as an example, suppose we want to investigate how much improvement we can gain by removing the "sync" call: following COZ's idea, we should keep the length of "sync" unchanged and slow down the disk write, but this will automatically increase the length of "sync". For this reason, we do not find an accurate way to apply COZ to off-CPU events.

That said, we find on-CPU and off-CPU analysis are not completely orthogonal, so COZ can provide hints to off-CPU analysis in certain cases. For example, in the BlockGrace experiment, the first bottleneck (BlockGrace-1) is caused by the computation threads waiting for the main thread to perform initialization: while wPerf identifies this bottleneck as a knot consisting of the main thread and the computation threads, COZ identifies that the initialization code is worth optimizing. Both reports can motivate the user to parallelize the initialization phase. The second bottleneck (BlockGrace-2), however, is caused by load imbalance among worker threads. While wPerf identifies a knot again, which motivates us to improve load balance, COZ reports the code in the computation threads is worth optimizing, which is certainly correct but misses the opportunity to improve load balance. Lock contention (e.g., in Memcached and MySQL) is another example: COZ can identify that execution in a critical section is worth optimizing. In this case, an experienced programmer may guess that reducing contention with fine-grained locking may also help, but without additional information, such guess may be inaccurate because long execution in the critical section

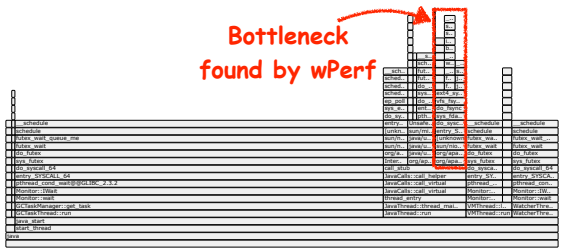


Figure 10: Off-CPU flame graph (ZooKeeper 0.1% write). The width of a box represents the length of the corresponding event.

can create a bottleneck as well even if there is almost no contention.

In summary, COZ can identify bottleneck on-CPU events, which wPerf cannot identify, but when regarding off-CPU events, COZ can at most provide some hints while wPerf can provide more accurate reports. Therefore, COZ and wPerf are mainly complementary.

Off-CPU flame graph. perf's off-CPU flame graph [57] can output all calls stacks causing waiting and aggregate them based on their lengths. However, it does not tell which events are important. One can focus on long events: for HBase, grep over NFS, and BlockGrace, the longest events happen to be the same as the ones reported by wPerf; for the others, the longest ones are not the same as the ones reported by wPerf, and such unimportant but long waiting are usually caused by threads waiting for some rare events, such as JVM's garbage collection threads or threads waiting for new connections. Figure 10 shows an example, in which one can see that the event reported by wPerf is not the longest one.

SyncPerf. SyncPerf [2] reports long or frequent lock contentions. For Memcached, it reports similar problems as wPerf, but for the other systems, it does not: for grep over NFS, SyncPerf does not report anything because grep does not have contention at all; for BlockGrace, SyncPerf reports asymmetric contention, but the key problem is imbalance among threads. We fail to run SyncPerf with MySQL, and the SyncPerf developers confirmed that it is probably because MySQL uses some functions SyncPerf does not fully implement. Note that the SyncPerf paper reported contentions in MySQL, so our problem may be caused by different versions of MySQL or glibc, etc. and if fixed, we believe SyncPerf and wPerf should identify similar bottlenecks.

5.3 Overhead

Table 3 reports overhead of wPerf. At runtime, wPerf incurs an average overhead of 5.1% for recording

	Slowdown	Trace size	Analysis
HBase	2.84%	1.4GB	110.6s
ZooKeeper	3.37%	393.9MB	23.8s
HDFS	3.40%	64.8MB	10.9s
NFS	0.77%	3.6MB	5.1s
BlockGrace	8.04%	110.7MB	14.7s
Memcached	2.43%	2.7GB	160.0s
MySQL	14.64%	7.4GB	271.9s

Table 3: Overhead of wPerf (recording for 90 seconds)

events. For BlockGrace and MySQL, the two applications with relatively large overhead, we further decouple the sources of their overhead: for BlockGrace, recording events in kernel and recording call stacks with perf incur 3.1% and 4.9% overhead respectively; for MySQL, recording events in kernel, recording call stacks with perf, and recording events in user space incur 0.5%, 4.2%, and 9.9% overhead respectively.

As shown in Table 3, the trace size and analysis time vary significantly depending on the number of waiting events in the application; the analysis time further depends on the number of nested waiting events. wPerf's parallel analysis helps significantly: for example, for HBase, with 32 threads, it reduces analysis time from 657.1 seconds to 110.6 seconds.

Besides, wPerf needs users' efforts to insert tracing functions for false wakeup and busy waiting events: we inserted 7 lines of code in HBase to trace false wakeup events and 12 lines of code in MySQL to trace busy waiting events; we do not modify the other applications since these two events are not significant in them.

6 Related Work

Performance analysis is a broad area: some works focus on identifying key factors to affect throughput [20, 58, 60] and others focus on latency-related factors [13, 33]; some works focus on a few abnormal events [7, 43, 45, 77] and others focus on factors that affect average performance [13, 20, 21, 37, 58, 60]. wPerf targets identifying key factors that affect the average throughput of the application. Therefore, this section mainly discusses related work in this sub-area.

As mentioned earlier, tools in this sub-area can be categorized into on-CPU analysis and off-CPU analysis.

On-CPU analysis. For single-threaded applications, traditional performance profilers measure the time spent in different call stacks and identify functions that consume most time. Following this idea, a number of performance profilers (e.g., perf [60], DTrace [20], oprofile [58], yourkit [74], gprof [25, 26], etc.) have been developed and applied in practice. Two approaches are widely used: the first is to periodically sample the call stack of the target application and use the number of samples spent in each function to approximate the time spent in each function; the second is to instrument the target application and trace certain function calls [44, 54].

For multi-threaded programs, a number of works try to identify the critical path of an algorithm [22, 30, 31, 48–50, 59, 66] and pieces of code that often do not execute in parallel [35, 38, 39]. COZ [16] can further estimate how much improvement we can gain by optimizing a certain piece of code, as discussed in Section 5.2.

Off-CPU analysis. To identify important waiting events, many existing tools (e.g., perf [60], yourkit [74], jprofiler [36], etc.) can rank waiting events based on their aggregated lengths. However, as shown in Section 2, long waiting events are not necessarily important.

A number of tools design metrics to identify important lock contentions [2, 8, 18, 24, 75]. For example, Freelunch [18] proposes a metric called "critical section pressure" to identify important locks; SyncProfiler [75] proposes a graph-based solution to rank critical sections; SyncPerf [2] considers both the frequency and length of contentions. However, they are not able to identify problems unrelated to contention.

SyncProf [75] and SyncPerf [2] can further identify the root cause of a problem and make suggestions about how to fix the problem. Similar as many other tools, wPerf does not provide such diagnosis functionality.

7 Conclusion and Future Work

To identify waiting events that limit the application's throughput, wPerf uses cascaded re-distribution to compute the local impact of a waiting event and uses wait-for graph to compute whether such impact can reach other threads. Our case studies show that wPerf can identify problems other tools cannot find. In the future, we plan to extend wPerf to distributed systems, by connecting wait-for graphs from different nodes.

Acknowledgements

Many thanks to our shepherd Cristiano Giuffrida and to the anonymous reviewers for their insightful comments. Manos Kapritsos provided invaluable feedbacks on early versions of this project. This material is based in part upon work supported by the NSF grant CNS-1566403.

References

[1] Event Tracing for Windows (ETW). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[2] ALAM, M. M. U., LIU, T., ZENG, G., AND MUZAHID, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 298–313.

[3] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 185–198.

[4] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 265–278.

[5] Apache HBASE. http://hbase.apache.org/.

[6] aync-profiler: Sampling CPU and HEAP Profiler for Java Featuring AsyncGetCallTrace and perf events. https://github.com/jvm-profiling-tools/async-profiler/releases.

[7] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 307–320.

[8] BACH, M. M., CHARNEY, M., COHN, R., DEMIKHOVSKY, E., DEVOR, T., HAZELWOOD, K., JALEEL, A., LUK, C.-K., LYONS, G., PATIL, H., AND TAL, A. Analyzing parallel programs with pin. *Computer 43*, 3 (Mar. 2010), 34–41.

[9] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 69–86.

[10] Blockgrace. https://github.com/wenleix/BlockGRACE.

[11] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[13] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 217–231.

[14] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 1–17.

[15] CloudLab. https://www.cloudlab.us.

[16] CURTSINGER, C., AND BERGER, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 184–197.

[17] D3.JS Javascript Library. https://d3js.org/.

[18] DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Continuously measuring critical section pressure with the free-lunch profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 291–307.

[19] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[20] Dtrace. http://dtrace.org/.

[21] ERLINGSSON, U., PEINADO, M., PETER, S., AND BUDIU, M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 311–326.

[22] GARCIA, S., JEON, D., LOUIE, C. M., AND TAYLOR, M. B. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 458–469.

[23] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[24] GOLDIN, M. Thread performance: Resource contention concurrency profiling in visual studio 2010. https://msdn.microsoft.com/en-us/magazine/ff714587.aspx.

[25] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[26] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1982), SIGPLAN '82, ACM, pp. 120–126.

[27] Apache HBase (TM) Configuration. http://hbase.apache.org/0.94/book/important_configurations.html.

[28] HBase Administration Cookbook. https://www.safaribooksonline.com/library/view/hbase-administration-cookbook/9781849517140/ch09s03.html.

[29] HDFS. http://hadoop.apache.org/hdfs.

[30] HE, Y., LEISERSON, C. E., AND LEISERSON, W. M. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2010), SPAA '10, ACM, pp. 145–156.

[31] HILL, J. M. D., JARVIS, S. A., SINIOLAKIS, C. J., AND VASILEV, V. P. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on* (Jan 1998), pp. 286–294.

[32] HOLT, R. C. Some deadlock properties of computer systems. *ACM Comput. Surv. 4*, 3 (Sept. 1972), 179–196.

[33] HUANG, J., MOZAFARI, B., AND WENISCH, T. F. Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 64–79.

[34] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[35] JOAO, J. A., SULEMAN, M. A., MUTLU, O., AND PATT, Y. N. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 223–234.

[36] Jprofiler. https://www.ej-technologies.com/products/jprofiler/overview.html.

[37] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., VENKATARAMAN, V., VEERARAGHAVAN, K., AND SONG, Y. J. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 34–50.

[38] KAMBADUR, M., TANG, K., AND KIM, M. A. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 452–463.

[39] KAMBADUR, M., TANG, K., AND KIM, M. A. Parashares: Finding the important basic blocks in multithreaded programs. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings* (Cham, 2014), F. Silva, I. Dutra, and V. Santos Costa, Eds., Springer International Publishing, pp. 75–86.

[40] KELLEY, J. E. Critical-path planning and scheduling: Mathematical basis. *Oper. Res. 9*, 3 (June 1961), 296–320.

[41] Kernel Probe. https://www.kernel.org/doc/Documentation/kprobes.txt.

[42] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 25–36.

[43] LI, J., CHEN, Y., LIU, H., LU, S., ZHANG, Y., GUNAWI, H. S., GU, X., LU, X., AND LI, D. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 7:1–7:14.

[44] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[45] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 378–393.

[46] Memaslap benchmark. http://docs.libmemcached.org/bin/memaslap.html.

[47] Memcached. http://memcached.org.

[48] MILLER, B. P., CLARK, M., HOLLINGSWORTH, J., KIERSTEAD, S., LIM, S. S., AND TORZEWSKI, T. Ips-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (Apr 1990), 206–217.

[49] MILLER, B. P., AND HOLLINGSWORTH, J. K. *Slack: A New Performance Metric for Parallel Programs*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[50] MILLER, B. P., AND YANG, C. IPS: an interactive and automatic performance measurement tool for parallel and distributed programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, September 1987* (1987), pp. 482–489.

[51] MySQL. http://www.mysql.com.

[52] Scaling the HDFS NameNode (part 2). https://community.hortonworks.com/articles/43839/scaling-the-hdfs-namenode-part-2.htmll.

[53] Hadoop Tuning Notes. https://anandnalya.com/2011/09/hadoop-tuning-note/.

[54] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[55] Network File System. https://en.wikipedia.org/wiki/Network_File_System.

[56] Off-CPU Analysis. http://www.brendangregg.com/offcpuanalysis.html.

[57] Off-CPU Flame Graphs. http://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html.

[58] OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net.

[59] OYAMA, Y., TAURA, K., AND YONEZAWA, A. Online computation of critical paths for multithreaded languages. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, UK, 2000), IPDPS '00, Springer-Verlag, pp. 301–313.

[60] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[61] perf-map-agent: A Java Agent to Generate Method Mappings to Use with the Linux 'perf' Tool. https://github.com/jvm-profiling-tools/perf-map-agent.

[62] RONG SHI, Y. G., AND WANG, Y. Evaluating scalability bottlenecks by workload extrapolation. In *26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOT '18)* (Milwaukee, WI, 2018), IEEE.

[63] SHVACHKO, K. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[64] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[65] SOARES, L., AND STUMM, M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 33–46.

[66] SZEBENYI, Z., WOLF, F., AND WYLIE, B. J. N. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Nov 2009), pp. 1–12.

[67] TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page. http://www.tpc.org/tpcc/.

[68] VALIANT, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[69] WANG, G., XIE, W., DEMERS, A. J., AND GEHRKE, J. Asynchronous large-scale graph processing made easy. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[70] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 495–509.

[71] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 279–294.

[72] XIE, W., WANG, G., BINDEL, D., DEMERS, A., AND GEHRKE, J. Fast iterative graph computation with block updates. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 2014–2025.

[73] XIONG, W., PARK, S., ZHANG, J., ZHOU, Y., AND MA, Z. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 163–176.

[74] Yourkit Java and .Net Profiler. https://www.yourkit.com/.

[75] YU, T., AND PRADEL, M. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 389–400.

[76] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 29–42.

[77] ZHAO, X., RODRIGUES, K., LUO, Y., YUAN, D., AND STUMM, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 603–618.

[78] ZHOU, F., GAN, Y., MA, S., AND WANG, Y. wPerf: Generic Off-CPU Analysis to Identify Inefficient Synchronization Patterns (technical report). https://web.cse.ohio-state.edu/oportal/tech_reports/9.

[79] Zookeeper. http://hadoop.apache.org/zookeeper.

Sledgehammer: Cluster-fueled debugging

Andi Quinn, Jason Flinn, and Michael Cafarella
University of Michigan

Abstract

Current debugging tools force developers to choose between power and interactivity. Interactive debuggers such as gdb let them quickly inspect application state and monitor execution, which is perfect for simple bugs. However, they are not powerful enough for complex bugs such as wild stores and synchronization errors where developers do not know which values to inspect or when to monitor the execution. So, developers add logging, insert timing measurements, and create functions that verify invariants. Then, they re-run applications with this instrumentation. These powerful tools are, unfortunately, not interactive; they can take minutes or hours to answer one question about a complex execution, and debugging involves asking and answering many such questions.

In this paper, we propose *cluster-fueled debugging*, which provides interactivity for powerful debugging tools by parallelizing their work across many cores in a cluster. At sufficient scale, developers can get answers to even detailed queries in a few seconds. Sledgehammer is a cluster-fueled debugger: it improves performance by timeslicing program execution, debug instrumentation, and analysis of results, and then executing each chunk of work on a separate core. Sledgehammer enables powerful, interactive debugging tools that are infeasible today. *Parallel retro-logging* allows developers to change their logging instrumentation and then quickly see what the new logging would have produced on a previous execution. *Continuous function evaluation* logically evaluates a function such as a data-structure integrity check at every point in a program's execution. *Retro-timing* allows fast performance analysis of a previous execution. With 1024 cores, Sledgehammer executes these tools hundreds of times faster than single-core execution while returning identical results.

1 Introduction

Debugging is onerous and time-consuming, comprising roughly half of all development time [24]. It involves detective work: using the tools at her disposal, a developer searches a program execution for clues about the root cause of correctness or performance problems.

Current debugging tools force developers to choose between power and interactivity. Tools such as gdb are interactive: developers can inspect program values, follow execution flow, and use watchpoints to monitor changes to specific locations. For many simple bugs, interactive debuggers like gdb allow developers to quickly identify root causes by asking and answering many low-level questions about a particular program execution.

Yet, complex bugs such as wild stores, synchronization errors, and other heisenbugs are notoriously hard to find. Consider a developer trying to uncover the root cause of non-deterministic data corruption in a Web server. She cannot use gdb because she does not yet know which values to inspect or which part of the server execution to monitor. So, she employs more heavyweight tools. She adds logging message and sprinkles functions to verify invariants or check data structures at various points in the server code.

Custom tools like logging and invariant checks are powerful, but they are definitely not interactive. First, the developer must execute a program long enough for a bug to occur. Complex bugs may not be evinced with a simple test case; e.g., rare heisenbugs may require lengthy stress testing before a single occurrence. Second, detailed logging and custom predicates slow down program execution, sometimes by an order of magnitude. This means that each new question requires a long wait until an answer is delivered, and diagnosing a root cause often requires asking many questions.

Ideally, our developer would have tools that are both powerful and interactive. Then, she could ask complex questions about her server execution and receive an answer in a few seconds. Yet, the tradeoff seems fundamental: these powerful tools are time-consuming precisely because they require substantial computation to answer complex questions about long program executions.

Cluster-fueled debugging solves this dilemma: it provides interactivity for complex tools by parallelizing their work across many cores in a compute cluster. With sufficient scale, developers see answers to even detailed queries in a few seconds, so they can quickly iterate to gather clues and identify a root cause.

Sledgehammer is the first general cluster-fueled debugger. It is designed to mirror current debugging workflows: i.e., adding logging [38] or invariant checks, recompiling, re-executing to reproduce the problem, and analyzing the output of the additional instrumentation. However, Sledgehammer produces results much faster

through parallelization of instrumentation and analysis. Like prior academic [17, 32, 34] and commercial [27, 31] tools, Sledgehammer is replay-based; i.e., it can deterministically reproduce any previously-recorded execution on demand for debugging. Replay facilitates iterative debugging because each question is answered by observing the same execution, ensuring consistent answers.

Sledgehammer uses deterministic replay for another purpose: it time-slices a recorded program execution into distinct chunks called *epochs*, and it runs each epoch on a different core. It uses `ptrace` to inject debugging code, called *tracers* into program execution. Vitally, Sledgehammer provides isolation so that tracers do not modify program behavior, guaranteeing that each replayed execution is consistent with the original recording. Because tracers are associated with specific points in the program execution and the execution is split across many cores, the overhead of both tracer execution and isolation is mitigated through parallelization.

Tracers may produce large amounts of data for complex debugging tasks, and processing this data could become a bottleneck. So, Sledgehammer also provides several options to parallelize data analysis. First, local analysis of each epoch can be performed on each core. Second, stream-based analysis allows information to be propagated from preceding epochs to subsequent epochs, allowing further refinement on each core. Finally, tree-based aggregation, terminating in a global analysis step, produces the final result.

Cluster-fueled debugging makes existing tools faster. Retro-logging [6, 15, 36] lets developers change logging in their code and see the output that would have been produced if the logging had been used with a previously-recorded execution. Retro-logging requires isolating modified logging code from the application to guarantee correct results. Both isolation and voluminous logging add considerable overhead. We introduce *parallel retro-logging*, which hides this overhead through cluster-fueled debugging to make retro-logging interactive.

Cluster-fueled debugging enables new, powerful debugging tools that were previously infeasible due to performance overhead. To demonstrate this, we have created *continuous function evaluation*, which lets developers define a function over the state of their execution that is logically evaluated after every instruction. The tool returns each line of code where the function return value changes. Continuous function evaluation mirrors the common debugging technique of adding functions that verify invariants or check data structure integrity at strategic locations in application code [9], but it frees developers from having to carefully identify such locations to balance performance overhead and the quality of information returned.

We have also created *parallel retro-timing*, which lets developers retroactively measure timing in a previously-recorded execution (a feature not available in prior replay-based debugging tools). Sledgehammer returns timing measurements as a range that specifies minimum and maximum values that could have been returned during the original execution.

This paper makes the following contributions:

- We present a general framework for parallelizing complex debugging tasks across a compute cluster to make them interactive.
- Parallelization makes scalability a first-class design constraint for debugging tools, and we explore the implications of this constraint.
- We introduce continuous function evaluation as a new, powerful debugging tool made feasible by Sledgehammer parallelization and careful use of compiler instrumentation and memory protections.
- We explore the fundamental limits of parallelization and show how to alleviate the bottlenecks experienced when trying to scale debugging.

We evaluate Sledgehammer with seven scenarios debugging common problems in memcached, MongoDB, nginx, and Apache. With 1024 compute nodes, Sledgehammer returns the same results as sequential debugging, but parallelization lets it return answers 416 times faster on average. This makes very complex debugging tasks interactive.

2 Usage

To use Sledgehammer, a developer records the execution of a program with suspect behavior for later deterministic replay. Recording could occur during testing or while reproducing a customer problem in-house. Deterministic replay enables parallelization. It also makes results from successive replays consistent, since each replay of the application executes the same instructions and produces the same values on every replay.

Next, the developer specifies a debugging query by adding *tracers* to the application source code. A tracer can be any function that observes execution state and produces output. Examples of tracers are logging functions, functions that check invariants, and functions for measuring timing. A tracer can be inserted at a single code location, inserted at multiple locations, or evaluated continuously. Thus, tracers are added in much the same way that developers currently add logging messages or invariant checks to their code.

A developer can also add *analyzers* to aggregate tracer output and produce the final result; e.g., an analyzer could filter log messages or correlate events to identify use-after-free bugs. Sledgehammer provides several ways to parallelize analysis. Developers can write local analyzers that operate only on output from one epoch of

program execution, stream analyzers that propagate data between epochs in the order of program execution, and tree-based analyzers that combine per-epoch results to generate the final result over the entire execution.

In summary, the interface to Sledgehammer is designed to be equivalent to the current practice of adding logging/tracing code and writing analysis code to process that output. However, Sledgehammer uses a compute cluster to parallelize application execution, instrumentation, and analysis, and, in our setup, produces answers in a few seconds, instead of minutes or hours.

3 Debugging tools

We have created three new parallel debugging tools.

3.1 Parallel retro-logging

Retro-logging [6, 15, 36] lets developers modify application logging code and observe what output would have been generated had that logging been used during a previously-recorded execution. We implement parallel retro-logging by adding tracers to the application code that insert new log messages; often tracers use the existing logging code in the application with new variables. Log messages are deleted via filtering during analysis, and log messages are modified by both inserting a new log message and filtering out old logging.

Parallelizing retro-logging has several benefits. First, the application being logged may run for a long time, and verbose logging causes substantial performance overhead. Second, even carefully-written logging code perturbs the state of the application in subtle ways, e.g., by modifying memory buffers and advancing file pointers. If left unchecked, these subtle differences cause the replayed execution to diverge from the original, which can prevent the replayed execution from completing or silently corrupt the log output with incorrect values. Isolation is required for correctness, and the cost of isolation is high. This cost is not unique to Sledgehammer: tools such as Pin [22] and Valgrind [26] that also isolate debugging code from the application have high overhead. Sledgehammer hides this overhead via parallelization.

3.2 Continuous function evaluation

Continuous function evaluation logically evaluates the output of a specified function after every instruction. It reports the output of the function each time the output changes and the associated instruction that caused the change. Continuous function evaluation can be used to check data structure invariants or other program properties throughout a recorded execution.

Actually evaluating the function after each instruction would be prohibitively expensive, even with parallelization. Sledgehammer uses static analysis to detect values read by the function that may affect its output and memory page protections to detect when those values change. This reduces performance overhead to the point where parallelization can make this debugging tool interactive.

3.3 Retro-timing

Many debugging tasks require developers to understand the timing of events within an execution. Replay debugging recreates the order of events, but not event timing. Thus, a recorded execution is often useless for understanding timing bugs.

Sledgehammer systematically captures timing data while recording an execution. To reduce overhead of frequent time measurements, it integrates time recording with the existing functionality for recording nondeterministic program events. When debugging, developers call `RetroTime`, a Sledgehammer provided function that returns bounds on the clock value that would have been read during the original execution. These bounds are determined by finding the closest time measurements in the replay log.

4 Scenarios

We next describe seven scenarios that show how Sledgehammer aids debugging. We use these scenarios as running examples throughout the paper and measure them in our evaluation.

4.1 Atomicity Violation

Concurrency errors such as atomicity violations are notoriously difficult to find and debug [21]. In this scenario, a memcached developer finds an error message in memcached's production log indicating an inconsistency in an internal cache. Memcached uses parallel arrays, `heads`, `tails` and `sizes`, to manage items within the cache. For each index, `heads[i]` and `tails[i]` point to the head and tail of a doubly-linked list, and `size[i]` holds the number of list items.

To use Sledgehammer, the developer first records an execution of memcached that exhibits the bug. Next, she decides to use continuous function evaluation and writes tracers to identify the root cause of the bug. To illustrate this process, we used existing assert statements in the memcached code to write the sample tracer in Figure 1. The `is_corrupt` function validates the correctness of a single list. The `check_all_lists` function returns "1" if any list is corrupt and "0" otherwise.

By adding `SH_Continuous(check_all_lists)` to the memcached source, the developer specifies that `check_all_lists` should be evaluated continuously. This outputs a line whenever the state of the lists transitions from valid to invalid, or vice versa. The `CFE_RETURN` macro prepends to each line the thread id and instruction pointer where the transition occurred.

The developer then writes an analysis function; we

```
1  bool is_corrupt (item *head, item *tail, int size) {
2    int count = 0;
3
4    while (tail->prev != NULL)
5      tail = tail->prev;
6    if (tail != head) return true;
7
8    while (head != NULL) {
9      head = head->next;
10     count++;
11   }
12   return (count != size);
13 }
14
15 char *check_ all_lists () {
16   for (int i = 0; i < SIZE; ++i)
17     if (is_corrupt (heads[i], tails[i], sizes[i]))
18       CFE_RETURN (``1'');
19   CFE_RETURN (``0'');
20 }
```

Figure 1: Tracer for the first memcached query.

```
1  void analyze (int in, int out) {
2    FILE *inf = fdopen(in), outf = fdopen(out);
3    map<int, int> invalid_count;
4    char line[128];
5    int location, tid, count;
6
7    while (getline(&line, NULL) > 0) {
8      sscanf(``%x:%x:%x\n'', &location, &tid, &count);
9      if (count) invalid[location] += count;
10   }
11   for (auto &it : invalid)
12     fprintf(outf, ``%x:%x\n'', it.first, it.second);
13 }
```

Figure 2: Analyzer for the first memcached query.

show the function she would write in Figure 2. This function reports all code locations where a transition to invalid occurs. We wrote this function so that the same code can be used for local and tree analysis.

Running this query doesn't reveal the root cause of the bug, as each transition to invalid occurs at a code location that is supposed to update the cache data structures. So, the developer next suspects a concurrency bug. The cache is updated in parallel; for each index `i`, a lock, `locks[i]`, should be held when updating the parallel arrays at index `i`. Thus, there are two invariants that should be upheld: whenever the arrays at index i become invalid, `locks[i]` should be held, and whenever `locks[i]` is released, the arrays should be valid.

```
1  char* check_all_lists () {
2    for (int i = 0; i < SIZE; ++i)
3      if (check(heads[i], tails[i], sizes[i]))
4        CFE_APPEND (``invalid:%x\n'', locks[i]);
5      else
6        CFE_APPEND (``valid:%x\n'', locks[i]);
7    CFE_RETURN();
8  }
9
10 void hook_lock (pthread_mutex_t *mutex) {
11   tracerLog(``0:%x:lock:%x\n'', tracerGettid(), mutex);
12 }
13
14 void hook_unlock (pthread_mutex_t *mutex) {
15   tracerLog(``0:%x:unlock:%x\n'', tracerGettid(), mutex);
16 }
```

Figure 3: Tracer for the second memcached query.

Figure 3 shows how the developer would modify the tracer for a second query. The `check_all_lists` function now appends the validity and lock for each item in the list to a string and returns the result. The developer also adds two functions that report when cache locks are acquired and released. She adds two more statements to the memcached source code to specify that these functions should run on each call to `pthread_mutex_lock` and `pthread_mutex_unlock`.

Figure 4 shows the new analysis routine that the developer would write. The analyzer is structured like a state-machine; each line of input is a transition from one state to the next. `lockset` tracks the locks currently held and `needed_locks` tracks which locks must be held until lists are made valid again. Line 14 checks the first invariant mentioned above, and line 28 checks the second.

We again use the same analyzer for both local and tree-based analysis. Since local analysis occurs in parallel, a needed lock may have been acquired in a prior epoch, and locks held at the end of an epoch may be needed in a future epoch. Thus, the analyzer outputs all transitions that it can not prove to be correct based on local information, as well as information that may be needed to prove transitions in subsequent epochs correct. The global analyzer at the root of the tree has all information, so any transition it outputs is incorrect.

In our setup, the query returns in a few seconds and identifies two instructions where an array becomes invalid while the lock is not held. One occurs during initialization (and is correct because the data structure is not yet shared). The other is the atomicity bug.

4.2 Apache 45605

In this previously reported bug [3], a Apache developer noted that an assertion failed during stress testing. The assertion indicated that a thread pushed too many items onto a shared queue. Without Sledgehammer, developers spent more than two months resolving the bug. They even proposed an incorrect patch, suggesting that they struggled to understand the root cause.

Four unsigned integers, `nelts`, `bounds`, `idlers` and `max_idlers`, control when items are pushed onto the queue. By design, `nelts` should always be less than `bounds`, and `idlers` should always be less than `max_idlers`. We emulated a developer using Sledgehammer to debug this problem by writing a tracer that

```
1  void analyzer (int in, int out) {
2   FILE *inf = fdopen (in), outf = fdopen (out);
3   map<int, set<int>> lockset;
4   map<int, set<int>> needed_locks;
5   char line[128], type[8];
6   int thread, ip, lock;
7
8   while (getline (&line, NULL) > 0) {
9    sscanf (``%x:%x:%s:%x'', ip, thread, type, lock);
10
11   if (!strcmp (type, ``lock''))
12    lockset[thread].insert (lock);
13   } else if (!strcmp(type, ``invalid'')) {
14    if (lockset[thread].contains (lock))
15     needed_locks[thread].insert (lock);
16    else
17     fprintf (outf, line);
18   } else if (!strcmp (type, ``valid'')) {
19    if (needed_locks[thread].contains (lock))
20     needed_locks[thread].remove (lock);
21    else
22     fprintf (out, line);
23   } else if (!strcmp (type, ``unlock'') {
24    if (lockset[thread].contains (lock))
25     lockset[thread].remove (lock);
26    else
27     fprintf (outf, ``%s'', line);
28    if (needed_locks[thread].contains (lock))
29     fprintf (outf, ``BUG: atomicity violation: %x\n'', ip);
30   } else {
31    fprintf(outf, ``%s'', line);
32   }
33  }
34
35  for (const auto &l_set : lockset)
36   for (lock : l_set.second)
37    fprintf (outf, ``lock:%x:%x\n'', l_set.first, lock);
38  for (const auto &l_set : needed_locks)
39   for (lock : l_set.second)
40    fprintf (outf, ``invalid:%x:0:%x\n'', l_set.first, lock);
41 }
```

Figure 4: Analyzer for the second memcached query.

uses continuous function evaluation to check these relationships and an analyzer that lists instructions that cause a relationship to no longer hold.

The query returns a single instruction that decrements `idlers`. As this result is surprising, we modified the query to also output the value of each integer when the transition occurs. This shows that the faulty instruction causes an underflow by decrementing `idlers` from 0 to `UINT_MAX`. From this information, the developer can realize that `idlers` should never be 0 when the instruction is executed and that the root cause is that the preceding `if` statement should be a `while` statement.

4.3 Apache 25520

In another previously-reported bug [2], an Apache developer found that the server log was corrupted after stress testing. Apache uses an in-memory buffer to store log messages and flushes it to disk when full. With Sledgehammer, a developer could debug this issue by writing a tracer that uses continuous function evaluation to validate the format of log messages in the buffer. The analyzer identifies instructions that transition from a correctly-formatted buffer to an incorrectly-formatted one. Running this query returns only an instruction that updates the size of the buffer after data has been copied into the buffer. This indicates that the buffer corruption occurs during the data copy before the size is updated.

Thus, the developer next writes a query to detect such corruption by validating the following invariant: each byte in the buffer should be written no more than once between flushes of the buffer to disk. The continuous function evaluation tracer returns a checksum of the entire buffer region (so that all writes to the buffer region are detected irrespective of the value of the size variable) and the memory address triggering the tracer (see `tracerTriggerMemory()` in Section 5.2). The developer also writes a tracer that hooks calls to the buffer flush function. The analyzer outputs when multiple writes to the same address occur between flushes. The output shows that such writes come from different threads, identifying a concurrency issue in which the instructions write to the buffer without synchronization.

4.4 Data corruption

Memory corruption is a common source of software bugs [20] that are complex to troubleshoot; often, the first step in debugging is reproducing the problem with more verbose logging enabled. In this scenario, an nginx developer learns that the server very infrequently reports corrupt HTTP headers during stress testing, even though no incoming requests have corrupt headers. Without Sledgehammer, he would enable verbose logging and run the server for a long time to try to produce a similar error. Reproduction is painful; verbose logging adds considerable slowdown and produces gigabytes of data.

With Sledgehammer, the developer uses parallel retro-logging to enable the most verbose existing nginx logging level over the failed execution recorded during testing. In nginx, this requires adding tracepoints in two dedicated logging functions. Each tracer calls a low-level nginx log function after specifying the desired level of verbosity. The developer also filters by regular expression to only collect log messages pertaining to HTML header processing. The same filter code can be run as a local analyzer without modification, so parallelization is trivial. In our setup, the Sledgehammer query returns results in a few seconds, and the developer notes that corruption occurs between two log messages. This provides a valuable clue, but the developer must iteratively add more logging to narrow down the problem. Fortunately, these messages can be added retroactively to the same execution; the resulting output is seen in a few seconds.

4.5 Wild store

Wild stores, i.e., stores to invalid addresses that corrupt memory, are another common class of errors that are reportedly hard to debug [20]. In this scenario, Mongodb crashes and reports an error due to a corruption in its key B-tree data structure. Mongodb has an existing debugging function that walks the B-tree and checks its validity. Without Sledgehammer, the developer must sprinkle calls to this function throughout the code, re-run the application to reproduce the rare error, and try to catch the corruption as it happens. Unfortunately, the corruption was introduced during processing of a much earlier request and lay dormant for over 10 seconds. Further, the wild store was performed by an unrelated thread, so it takes numerous guesses and many iterations of running the program to find the bug.

With Sledgehammer, the developer specifies that the existing Mongodb debugging function should be evaluated continuously. Since the B-tree is constantly being modified, its validity changes often in the code that adds and deletes elements. The developer therefore writes a simple analyzer that counts the number of transitions that occur at each static instruction address. The same code is used for both local and tree-based analyzers. The query returns in under a minute. It reports three code locations where the data structure becomes invalid exactly once: two are initialization and the third is the wild store.

4.6 Memory leak

Memory leaks, double frees, and use-after-free bugs require reasoning about an execution's pattern of allocations and deallocations. In this scenario, an nginx developer notes that a large code change has introduced very infrequent memory leaks that lead to excessive memory usage for long-running servers. One option for tracking down this bug is to run a tool like Valgrind [26] over a long execution with varied requests. Due to the overhead of Valgrind instrumentation, this takes many minutes to return a result over even a relatively short execution.

With Sledgehammer, the developer adds three tracers and hooks the entry and exit of routines such as `malloc` and `free`. The analyzer matches allocations and deallocations and reports remaining unallocated memory. Parallelizing the analyzer is straightforward: the sequential analyzer can be used for tree-based and local analysis without modification. Stream analysis requires adding 10 lines of code to pass a list of allocated memory regions that have not yet been deallocated from epoch to epoch. In our setup, a Sledgehammer query identifies leaked memory in nginx in a few seconds.

4.7 Lock Contention

Rare performance anomalies are hard to debug. One common source of performance anomalies is lock contention [33]. Low-level timing data is informative, but gathering such data has high overhead and may prevent the anomaly from occurring. In this scenario, a memcached developer sees infrequent requests that have much longer latencies than expected. She runs a profiler, but the tool reports only average behavior, which obscures the occasional outlier. So, she sprinkles timing measurements at key points in her code and re-runs the application many times to drill down to the root cause: lock contention with a background thread. Each run requires a long time to exhibit an anomaly and difficult analysis to determine which requests are outliers in each new execution.

With Sledgehammer, the developer runs a query that gathers `RetroTime` data at key points in request parsing, starting with existing timing code originally disabled during recording. Because queries are fast, she retroactively adds even more timing code, and she can iterate quickly to drill down to the suspect lock acquisition. Her analysis function tracks time taken in each specified request phase, and compares breakdowns for the five longest requests with average behavior. A final query identifies the thread holding the contended lock by combining retro-timing with tracers that hook mutex acquisition and release. Analysis of the tracelog identifies the background thread that holds the lock on which the anomalous requests wait.

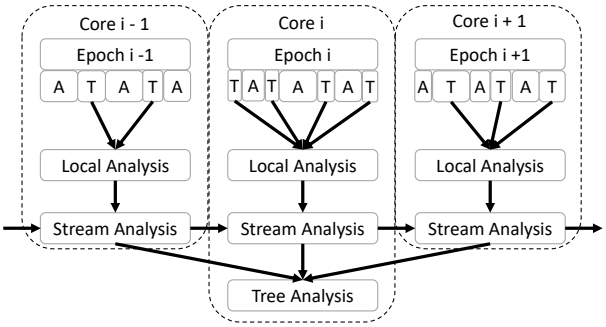


Figure 5: Sledgehammer architecture overview. A replay is divided into n epochs. Each core runs an epoch by executing the original application (A boxes) and injecting tracers (T boxes). Local analysis runs on each core with output from a single epoch, stream analysis takes input from previous epochs and sends output to subsequent ones. Tree analyzers combine output from multiple epochs.

5 Design and implementation

Figure 5 shows how Sledgehammer parallelizes debugging. The developer specifies (1) a previously-recorded execution to debug, (2) tracers that run during a replay of that execution, and optionally, (3) analysis functions that aggregate tracer output to produce a final result. Sledgehammer parallelizes the replayed execution, tracers, and analysis across many cores in a cluster.

Section 5.2 discusses how developers specify tracers by annotating their source code to add logging and instrumentation. Sledgehammer parses the source code to extract the tracers, the arguments passed to each tracer, and the locations where tracers should be invoked.

Section 5.3 describes how Sledgehammer prepares for query execution by distributing generic (non-query-specific) information needed to replay the execution to available compute nodes. It divides the execution into epochs of roughly-equal duration, where the number of epochs is determined by the number of cores available. Read-only data shared across epochs, e.g., the replay log, application binaries, and shared libraries, are read from a distributed file system. Sledgehammer caches these files on local-disk for improved performance on subsequent queries. The per-epoch state, e.g., application checkpoints at the beginning of each epoch, is generated in parallel, with each core generating its own state.

As discussed in Section 5.4, Sledgehammer runs a query by executing each epoch in parallel on a separate core. Epoch execution starts from a checkpoint and replays non-deterministic operations from the replay log to reproduce the recorded execution. Sledgehammer uses `ptrace` to insert software breakpoints at code locations where tracers should run. When a breakpoint is triggered, it runs the tracer in an isolated environment that rolls back any perturbation to application state after the tracer finishes. To support continuous function evaluation, Sledgehammer uses page protections to monitor memory addresses that may affect the return value of the function; it triggers a tracer when one of those addresses is updated.

Section 5.5 discusses how analyzers process the stream of output from tracers. As shown in Figure 5, Sledgehammer supports three types of analysis routines: local, stream, and tree. Local analysis (e.g., filtering) operates on tracer output from a single epoch. Stream analysis allows information to be propagated from epochs earlier in the application execution to epochs later in the execution. Sledgehammer runs a stream analyzer on each compute node; each analyzer has sockets for reading data from its predecessor epoch and sending data to its successor. A tree analyzer combines input from many epochs and writes its output to `stdout`. For a large number of cores, these analyzers are structured as a tree with the root of the tree producing the final answer to the query. Thus, a purely sequential analysis routine can always run as the root tree analyzer.

5.1 Background: Deterministic record and replay

Sledgehammer uses deterministic record and replay both to parallelize the execution of a program for debugging, and also to ensure that successive queries made by a developer return consistent results. Deterministic replay [11] allows the execution of a program to be recorded and later reproduced faithfully on demand. During recording, all inputs from nondeterministic actions are written to a *replay log*; these values are supplied during subsequent replays instead of performing the nondeterministic operations again. Thus, the program starts in the same state, executes the same instructions on the same data values, and generates the same results.

Epoch parallelism [35] is a general technique for using deterministic replay to partition a fundamentally sequential execution into distinct epochs and then execute each epoch in parallel, typically on a different core or machine. Determinism guarantees that the result of stitching together all epochs is equivalent to a sequential execution of the program. Replay also allows an execution recorded on one machine to be replayed on a different machine. There are few external dependencies, since interactions with the operating system and other external entities are nondeterministic and replayed from the log.

Sledgehammer uses Arnold [10] for deterministic record and replay due to its low overhead (less than 10% for most workloads) and because Arnold supports epoch parallelism [29]. We modified Arnold to support ptrace-aware replay, in which Sledgehammer sets breakpoints and catches signals. We also modified Arnold to run tracers in an isolated environment where they can allocate memory, open files, generate output, etc. Our modifications roll back the effects of these actions after the tracer finishes to guarantee that the replay of the original execution is not perturbed, similar to prior systems that support inspection of replayed executions [6, 15, 16]. In other words, the same application instructions are executed on the same program values, but Sledgehammer inserts additional tracer execution into replay and the instrumentation needed to support that execution. We also modified Arnold to capture additional timing data during recording to support retro-timing.

5.2 Sledgehammer API

Developers debug a replayed execution by specifying tracers that observe the program execution, defining when those tracers should execute, and supplying analysis functions that aggregate tracer output. This is analogous to placing log functions in source code and writing programs to process log output.

5.2.1 Tracers

Tracers are functions that execute in the address space of the program being debugged, allowing them to observe the state of the execution. Tracers are compiled into a shared library that is loaded dynamically during query execution. Tracers write output to a logging stream called the *tracelog*; this output is sent to analysis routines for aggregation. Tracers can write to the tracelog directly by calling a Sledgehammer-supplied function or they can

specify that all output from a specific set of file descriptors should be sent to the tracelog.

Isolation Tracers must not perturb program state. Even a subtle change to application memory or kernel state can cause the replay to diverge from the recording, leading to replay failure, or even worse, silent errors introduced into the debugging output. None of the queries in our scenarios run without isolation. As Section 5.4.1 describes, Sledgehammer isolates tracers in a sandbox during execution; any changes to application state are rolled back on tracer completion. Sledgehammer has two methods of isolation with different tradeoffs between performance and code-generality: fork-based and compiler-based.

With fork-based isolation, tracers run as separate processes. Developers have great flexibility. A tracer can make arbitrary modifications to the program address space, and it can make system calls that write to the tracelog or that affect only child process state. A tracer may link to any application code or libraries and invoke arbitrary functionality within that code, provided it does not make system calls that externalize state. However, we found that fork-based isolation was very slow to use with frequently-executed tracers.

Thus, we added support for compiler-based isolation, in which tracers can execute more limited functionality. This isolation is enabled by compiling tracers with a custom LLVM [18] pass. Tracers can modify any application memory or register. However, they must use a Sledgehammer-provided library to make system calls. This library prevents these calls from perturbing application state. A tracer may call functions in application code or libraries only if that code is linked into the tracer and compiled with LLVM. Since LLVM cannot compile glibc by default, Sledgehammer provides many low-level functions for tracer usage. Our compiler pass verifies that all functions linked into a tracer call only other functions compiled with the tracer or Sledgehammer library functions. Our results in Section 6.5 show that compiler-based isolation executes queries 1–2 orders of magnitude faster than fork-based isolation.

The tracestore Tracers must execute independently. Since tracers run in parallel in different epochs, a tracer cannot rely on state or output produced by any tracer executed earlier in the program execution. Yet, there are often many tracers executed during a single epoch, and sharing data between them can be a useful optimization. For instance, it is wasteful for each tracer to independently determine the file descriptor used for logging by an application.

Sledgehammer provides a *tracestore* for opportunistic sharing of state within an epoch. If data in the tracestore is available, a tracer uses it; if not available, it obtains the data elsewhere. Sledgehammer allocates the tracestore by scanning the replay log to find an address region never allocated by the execution being debugged; it maps the tracestore into this region. This prevents tracestore data from perturbing application execution.

Sledgehammer initializes the tracestore at the beginning of each epoch, prior to executing any application instructions. The developer can supply an initialization routine that inspects application state and sets variables to initial values. If compiler-based isolation is being used, the LLVM compiler pass automatically places all static tracer function variables in the trace store and initializes them at the start of each epoch.

Tracers read and write tracestore values, and updates are propagated to all subsequent tracer executions until the end of the epoch. Tracers may dynamically allocate and deallocate memory in the tracestore; the memory remains allocated until the end of the epoch. All of our scenarios use the tracestore to cache file descriptors, which avoids the overhead of opening and closing files in each tracer. The continuous function evaluation scenarios also cache lists of memory addresses accessed by the function.

Tracer Library Sledgehammer provides several functions that implement common low-level tasks, including:

- `tracerTriggerAddress()`, which returns the instruction pointer that triggered the tracer.
- `tracerStack()`, which returns the stack pointer when the tracer was called.
- `tracerTriggerMemory()`, which returns the memory address that triggered a continuous function evaluation tracer.

5.2.2 Tracepoints

Sledgehammer inserts tracers at *tracepoints*, which are user-defined locations in the application being debugged. There are several ways to add tracepoints. First, a location-based tracepoint executes a tracer each time the program execution reaches a given location. Our data corruption scenario uses this method to add tracers to nginx log routines. These tracepoints are specified by adding annotations to the application source code at the desired locations.

Second, a user can *hook* a specific function to invoke a tracer each time a given function is called or whenever a function exits. The tracer receives all arguments passed to the function by default. For example, the memory leak scenario hooks the entry and exit of `malloc` and `free` to track memory usage.

Third, a continuous function evaluation logically inserts a tracepoint to evaluate the function after every program instruction. In practice, Sledgehammer tracks the values read by the function and uses memory page protection to detect when those values change. It only runs the function at these instances. Hooks and continuous

functions can be specified by annotations anywhere in the application source code since their effects are global to the entire execution.

When running a query, developers specify which C/C++ source files contain their modified source code. The Sledgehammer parser scans these files and extracts all tracepoint annotations. It correlates each tracer with a line or function name in the application source code as appropriate. Next, it uses the same method as gdb to convert source code lines and function symbols to instruction addresses. For each parameter passed to a tracepoint, the parser determines the location of the symbol, i.e., its memory address or register.

Developers who lack source code or use other programming languages can instead use gdb-like syntax to specify tracepoints, or they can specify all functions residing in a particular binary, or matching some regex. In this case, Sledgehammer leverages UNIX command-line utilities and gdb scripts to associate tracepoints with instruction pointers and symbols.

5.3 Preparing for debugging queries

Much of the work required to run a parallel debugging tool is query-independent: it can be done once, before running the first query, and reused for future queries. To prepare a recorded execution for debugging, a *master node* parses the replay log and splits the execution into distinct epochs, where the number of epochs is set to the number of cores available. Each core is assigned a distinct epoch. Currently, Sledgehammer requires each epoch to start and end on a system call. The master divides epochs so that each has approximately the same number of system calls in the replay log.

Next, the master distributes or creates the data needed to replay execution. Arnold replay requires a deterministic replay log, application binaries and libraries, and snapshots of any read-only files [10]. These files are read-only and accessed by many epochs, so the master places them in a distributed file system and sends a message to compute cores informing them of the location.

Each epoch starts at a different point in the program execution. Prior to instrumenting and running the epoch, Sledgehammer must re-create the application state at the beginning of the epoch. A simple approach would replay the application up to the beginning of the epoch. However, for the last epoch, this process takes roughly as long as the original execution of the program. To avoid this performance overhead, Sledgehammer starts each epoch from a unique checkpoint.

During recording, Sledgehammer takes periodic checkpoints every few seconds. This creates a relatively small set of checkpoints that are distributed to compute nodes by storing them in the distributed file system. Prior to running the query, the master asks each compute core to create an epoch-specific checkpoint. Each core starts executing the application from the closest previous recording checkpoint, pauses at the beginning of its epoch, and takes a new checkpoint. This process effectively parallelizes the work of creating hundreds or thousands of epoch-specific checkpoints, and it avoids having to store and transfer many large checkpoints.

Sledgehammer hides the cost of checkpoint creation in two ways. First, it overlaps per-epoch checkpoint creation with parsing of source code. Second, it caches checkpoints on each core so that they can be reused by subsequent queries over the same execution.

5.4 Running a parallel debugging tool

To run a query, the master sends a message to each compute core specifying the shared libraries that contain the compiled tracers and analysis functions. It also sends a list of tracepoints, each of which consists of an instruction address in the application being debugged, a tracer function, and arguments to pass to that function.

Upon receiving the query start message, a compute core restores its per-epoch checkpoint and loads the tracer dynamic library into the program address space via `dlopen`. Sledgehammer uses `dlsym` to get pointers to tracer functions. Unfortunately, the dynamic loader modifies program state and causes divergences in replay. Sledgehammer therefore checkpoints regions that will be modified before invoking the loader and restores the checkpointed values after the loader executes.

Prior to starting an epoch, each core also maps the tracestore into the application address space and calls the tracestore initialization routine. Each compute core starts a control process that uses the `ptrace` interface to manage the execution and isolation of tracer code. For each location-based tracepoint or function hook, the control process sets a corresponding software breakpoint at the specified instruction address by rewriting the binary code at that address with the `int 3` instruction.

Each core replays execution from the beginning of its epoch. When a software breakpoint is triggered, replay stops and the control process receives a `ptrace` signal. The control process rewrites the application binary to call the specified tracer with the given arguments. It uses one of the isolation mechanisms described next to ensure that the tracer does not perturb application state. After the tracer executes, the control process rewrites the binary to restore the software breakpoint.

5.4.1 Isolation

Tracer execution must be side-effect free: any perturbations to the state of the original execution due to tracer execution can cause the replay to diverge and fail to complete, or such perturbation can lead to incorrect debugging output. Sledgehammer supports fork-based and compiler-based isolation.

Fork-based isolation When a tracepoint is triggered, the control process forks the application process to clone its state. The parent waits until the child finishes executing. The control process rewrites the child's binary to call the tracer. As the tracer executes, it may call arbitrary code in the application and its libraries, but it must be single-threaded. The kernel sandboxes the system calls called by the child process. It allows system calls that are read-only or perturb only state local to the child process (e.g., its address space). To avoid deadlocks, Sledgehammer ignores synchronization operations made by the tracer; this is safe only because the tracer itself is single-threaded. The kernel also redirects output from any file descriptors specified by the developer to the tracelog; this is convenient for capturing unmodified log messages. Tracelog output can also be generated by system calls made by the Sledgehammer library. System calls that modify state external to the process (e.g., writing to sockets or sending signals) are disallowed. System calls that observe process state, e.g., getpid(), return results consistent with the original recording.

At the end of tracer execution, the child process exits and the tracer restarts application execution. If a tracer fails, the control process receives the signal via *ptrace* and resumes application execution.

Compiler-based isolation Our early results showed that fork-based isolation was often too slow for frequently-executed tracers. So, we created compiler-based isolation, which improves performance at the cost of losing some developer flexibility. With compiler-based isolation, tracer libraries must be self-contained; i.e., rather than calling application or library code from a tracer, that code must be copied or compiled into the tracer library. This means that tracers must use a set of standard library functions provided by Sledgehammer instead of calling those functions directly. Tracers must also be single-threaded and written in C/C++.

Sledgehammer compiles tracers with LLVM. A custom compiler pass inserts code into the tracer that instruments all store instructions and dynamically logs the memory locations modified by tracer execution and the original values at those locations to an undo log. The compiler pass inserts code before the tracer returns that restores the original values from the undo log. It also checkpoints register state before executing a tracer and restores that state on return. To avoid deadlocks, the compiler pass omits any synchronization instructions in the tracer; this is safe only because the tracer itself is single-threaded and all its effects are rolled back. The compiler pass verifies that the tracer is self-contained; e.g., that it does not make any system calls.

We noticed that most addresses in tracer undo logs were stack locations. Rather than log all of these stores, Sledgehammer allocates a separate stack for tracer execution and switches the stack pointer at the beginning and end of tracer execution. The compiler pass statically determines instructions that write to the stack via an intra-procedural points-to analysis, and it omits these stores from the undo log. Some variables are passed to the tracer on the stack, so Sledgehammer explicitly copies this data when switching stacks.

If a tracer fails, the control process catches the signal, runs the code to undo memory modifications, restores register state, and continues the application execution.

5.4.2 Support for continuous function evaluation

Continuous function evaluation must use compiler-based isolation. When a tracer runs, the compiler pass tracks the set of memory addresses read. The tracer is guaranteed to be deterministic because it cannot call non-deterministic system calls and must be single-threaded. Therefore, the value produced by the tracer cannot change unless one of the values that it has read changes.

Sledgehammer uses memory page protections to detect if any value read by a tracer changes. The control process causes the continuous function to be evaluated at the beginning of the epoch, before any application instruction executes. Tracer execution generates an initial set of addresses to monitor; the compiler adds instrumentation to record this *monitor set* in the tracestore. Sledgehammer executes the tracer only to initialize the monitor set, so tracer output is not logged to the tracelog. The control process asks the kernel to mark all pages containing at least one address in the monitor set as read-only.

When a page fault occurs due to the application writing to one of these pages, the kernel alerts the control process. The control process unprotects the page and single-steps the application. Then, the control process checks if the the faulting address is in the monitor set. If the address is not in the set, the page fault is due to false sharing, so Sledgehammer re-protects the page and continues execution.

If the address is in the monitor set, Sledgehammer runs the tracer again, and records its output in the tracelog. If the tracer faults on a page in the monitor set, the control process unprotects the page and resumes execution of the tracer. Since tracers do not write to many of the pages in the monitor set, unprotecting on demand is much more efficient than unprotecting all pages before tracer execution.

After the tracer completes, Sledgehammer updates the monitor set. If a page is added to the monitor set, Sledgehammer protects it. However, if a page is removed from the monitor set, Sledgehammer does not unprotect the page until the next page fault; this optimization improves performance by deferring work.

The stack switching optimization used for compiler-based isolation is also useful for continuous function

evaluation. Reads of addresses on the stack are detected via an intra-procedural points-to analysis and not instrumented. Any remaining stack reads are detected dynamically from their addresses.

5.4.3 Support for retro-timing

To support retro-timing, we modified Arnold to query the system time when a replay event occurs: such events include all system calls, signals delivered, and synchronization operations, including low-level synchronization in glibc. The timing information is written into the replay log for efficiency. Since Arnold is already paying the cost of interrupting the application and logging its activity, the additional performance cost of querying the system time is minimal (1%, as measured in Section 6.6).

A typical replay log will have tens of millions of events even for a few seconds of execution. Logging all this data would introduce substantial slowdown, so we compress the timing data by only logging the time if the difference from the last logged time is greater than 1 us.

Sledgehammer provides a library function to query time retroactively. Starting from the application's current location in the replay log, Sledgehammer finds and returns the immediately preceding and succeeding time recorded in the log. Reading the clock at this point in the execution would have returned a value in this range.

5.5 Aggregating results

Tracelog output can be quite large, so Sledgehammer allows developers to write analysis routines that aggregate the tracelog data. It provides several options for parallelizing analysis to improve performance.

There are three types of analysis routines. A *local analyzer* runs on each compute core and operates only over the tracelog data produced by a single epoch. For example, the data corruption scenario uses a local analyzer to filter undesired messages from verbose logging. If a local analyzer is specified, Sledgehammer creates an analysis process that loads and executes the local analyzer from a dynamic library. Local analyzers receive tracelog data on an input file descriptor and write to an output file descriptor. Sledgehammer uses shared memory to implement high-performance data sharing.

A *stream analyzer* passes information from epoch to epoch along the direction of program execution. The memory leak scenario passes allocated chunks of memory to succeeding epochs so that they can be matched with corresponding frees. This allows each core to reduce the amount of output data it produces.

Each epoch's stream analyzer has an input file descriptor on which it receives the output of the local analyzer (or the tracelog data if no local analyzer is being used). The stream analyzer has an additional file descriptor on which it receives data from its predecessor epoch. It has two output file descriptors: one to which it writes analysis output and another by which it passes data to its successor epoch. Data is passed between epochs via TCP/IP sockets. Each stream analyzer closes the output socket when it is done passing data to its successor, and each learns that no more data will be forthcoming by observing that the input socket has been closed.

A *tree analyzer* combines the output of many epochs. Each compute core sends its output to the node running the tree analyzer via a TCP/IP socket. Sledgehammer receives the data, buffers and reorders the data, then passes the output of the prior stage to the tree analyzer in the order of program execution. The tree analyzer aggregates the data and writes its output to a file descriptor.

By default, Sledgehammer allows a tree analyzer to combine up to 64 input streams. Therefore, if there are less than 64 epochs, a single tree analyzer performs a global aggregation.

Since use of these analyzers is optional, the simplest form of aggregation is NULL tree aggregation, in which Sledgehammer concatenates all tracelog output into a file in order of application execution. Alternatively, a developer may take any existing sequential analysis routine and run it as a tree analyzer at the root of the tree. However, Section 6.4 reports substantial performance benefits for many queries from using local, stream, and tree aggregation to parallelize analysis.

6 Evaluation

Our evaluation answers the following questions:

- How much does Sledgehammer reduce the time to get debugging results?
- What are the challenges for further scaling?
- What is the benefit of parallelizing analysis?
- Does compiler-based isolation reduce overhead?

6.1 Experimental Setup

We evaluated Sledgehammer using a CloudLab [30] cluster of 16 r320 machines (8-core Xeon E5-2450 2.1 GHz processors, 16 GB RAM, and 10 Gb NIC). Since several applications we evaluate use at least 2 GB of RAM, we only use 4 cores on each machine, yielding 64 total cores for parallelization. To investigate scaling, we emulate more cores by splitting the execution into 64-epoch subtrees, each with their own tree analyzer, and running the subtrees iteratively. We calculate the time for the final tree aggregation by distributing subtree outputs across the cores and measuring the time to send all outputs to a root node and run the global analyzer. We add this time to the maximum subtree execution time. This estimate is pessimistic since no output is sent until the last byte has been generated by the last tree analyzer. We also do not run stream analyzers beyond 64 cores.

Our results assume that the query-independent preparation of Section 5.3 (e.g., parallel checkpoint genera-

Benchmark	Application	Replay time(s)	Tracer calls (millions)	1 Core query time(s)	64 Cores query time(s)	64 Cores speedup	1024 Cores query time(s)	1024 Cores speedup
Data corruption	nginx	2.0	7.7	324.8 (±2.2)	7.2 (±0.2)	45 (±1.0)	1.0 (±0.0)	330 (±13)
Wild store	MongoDB	30.1	3.4	7688.4 (±13.5)	181.3 (±6.0)	42 (±2.0)	17.2 (±1.0)	446 (±15)
Atomicity violation	memcached	98.5	42.8	7852.4 (±20.1)	173.1 (±1.2)	45 (±0.3)	13.7 (±0.7)	573 (±15)
Memory leak	nginx	76.0	3.6	1575.2 (±8.1)	30.3 (±0.2)	52 (±0.3)	2.8 (±0.2)	559 (±25)
Lock contention	memcached	93.4	75.5	3281.8 (±17.5)	68.3 (±0.6)	48 (±0.5)	10.9 (±1.2)	301 (±16)
Apache 45605	Apache	50.7	1.9	249.9 (±1.1)	5.2 (±0.5)	48 (±0.5)	1.0 (±0.6)	255 (±15)
Apache 25520	Apache	60.1	3.7	717.3 (±1.6)	12.9 (±0.0)	55 (±0.2)	1.2 (±0.0)	601 (±4.4)

Table 1: Sledgehammer performance. This table shows how Sledgehammer speeds up the time to run a debug query with 64 and 1024 cores, as compared to sequential (1 core) execution. For reference, we also show the time to replay the application without debugging and the number of tracers executed during each query. Figures in parentheses are 95% confidence intervals.

tion) is already completed. Preparation is only done once for each execution and can be done in the background as the developer constructs a query. We measured this time to be proportional to the recording checkpoint frequency; e.g., preparation takes an average of 2.1 seconds when the record checkpoint interval is every 2 seconds.

6.2 Benchmarks

We reproduce the 7 scenarios described in Section 4 by injecting the described bug into each application and running the specified Sledgehammer query. In each scenario, our query correctly identifies the bug. All reported results are the mean of 5 trials; we show 95% confidence intervals. Queries use compiler-based isolation and parallelize analysis as described in each scenario. We use the following workloads:

- **Data corruption** We send nginx 100,000 static Web requests.
- **Wild store** We send MongoDB workload A from the YCSB benchmarking tool [7].
- **Atomicity violation** We use memtier [25] to send memcached 10,000 requests and execute the final query described in the scenario.
- **Memory leak** We send nginx 2 million static Web requests. By default, nginx leaks memory with this workload, so we did not inject a bug.
- **Lock contention** We use memtier [25] to send memcached 10,000 requests and execute the final query that hooks `pthread functions` and measures timing at 5 tracepoints.
- **Apache 45605** We recreate the bug by stress testing using scripts from a collection of concurrency bugs [37] and run the final query.
- **Apache 25520** We recreate the bug by stress testing Apache and run the final query.

6.3 Scalability

Table 1 shows results for the 7 scenarios. The first column shows the time to replay the execution with no debugging. The next column shows the number of tracers executed during the query. The remaining columns compare sequential (1 core) query time with performance at 64 and 1024 cores, respectively.

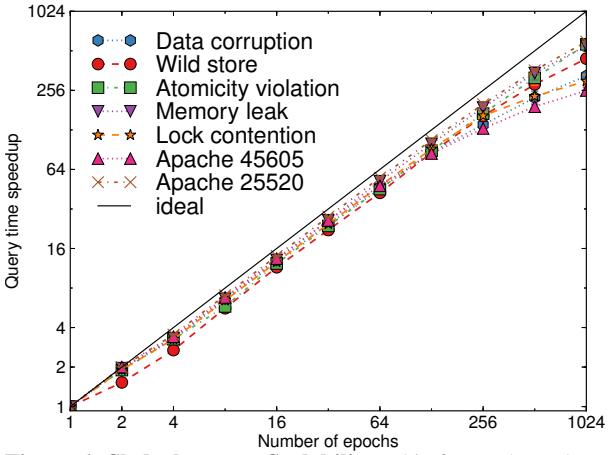


Figure 6: Sledgehammer Scalability. This figure shows how query time improves as the number of cores increases.

The wild store and atomicity violation scenarios take over 2 hours to return a result with sequential execution. The simplest scenario, Apache 45605, still takes over 4 minutes when executed sequentially. With 64 cores, Sledgehammer speeds up these queries by a factor of 42–55 (with a geometric mean of 48). With 1024 cores, the speedup is 255–601 with a mean of 416. Queries that take hours when executed sequentially return in less than 20 seconds. The data corruption and Apache 45605 queries returns results in one second. At 1024 cores, the result is returned faster than the time to replay the execution sequentially without debugging in all cases.

Figure 6 shows how Sledgehammer performance scales as the number of cores increases from 1 to 1024. The diagonal line through the origin shows ideal scaling. Most queries approach ideal scaling, and all continue to scale up to 1024 cores. However, some start to scale less well as the number of cores approaches 1024.

6.3.1 Scaling bottlenecks

We next investigated which factors hinder Sledgehammer scaling. One minor factor is disk contention. Arnold stores replay logs on local disk, which leads to contention when 4 large server applications each read their logs during epoch execution on separate cores. We measured this overhead as ranging from 0 to 41% at 4 cores per node, with an average of 15%. This accounts for some of the dip in scalability from 1 to 4 cores.

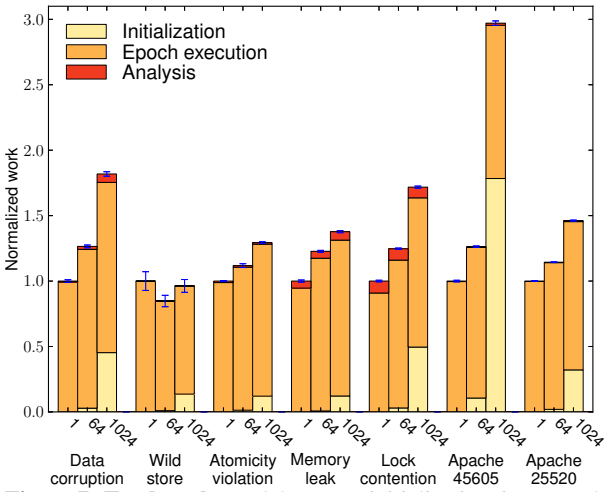


Figure 7: Total work. Each bar sums initialization time, epoch execution time, and analysis time overs all epochs. This shows how much extra work is created by parallelization.

The last step in tree analysis is sequential; its performance does not improve with the number of cores. At 1024 cores, this step is only 0.1–7% of total query time in our benchmarks. While it could be a factor for higher numbers of cores, it has little impact at 1024 cores.

For each query, Figure 7 totals individual execution time over all cores when using 64 and 1024 cores, normalized to execution time for single-core execution. Initialization includes restoring checkpoints and mapping tracers into the application address space. Epoch execution is the time to run the application and its tracers, Analysis includes all local, stream, and tree-based analysis. As expected, the cost of per-node initialization increases as the number of cores increases; this is especially noticeable in the Apache 45605, data corruption and lock contention scenarios. Initialization overhead is the primary factor inhibiting the scalability of Apache 45605. Initialization will eventually bound Sledgehammer scalability in other scenarios as well, but it is not the most important factor at 1024 cores.

Interestingly, the total work for the wild store scenario actually decreases slightly as we increase the number of cores. Continuous function evaluation defers work when pages are deleted from the monitor set. For shorter epochs, deleted pages are more likely to never be accessed again; work deferred is never done. At 1024 cores, this effect is dwarfed by increasing per-node initialization work, so total work increases again.

In most scenarios, the most significant barrier to scalability is workload skew. As Sledgehammer partitions epochs into smaller chunks, we see more imbalance in the work done by different epochs. Outlier epochs lead to high tail latency [8]. We quantify skew in Table 2 as the ratio of maximum epoch execution time over mean epoch execution time. Perfect partitioning would yield a skew of 1, but Sledgehammer sees average skew of 1.19 at 64 scores and 1.60 at 1024 cores. Skew is the most important factor in the decreased scaling seen in Figure 6.

	Skew	
Benchmark	64 cores	1024 cores
Data corruption	1.13 (±0.04)	1.72 (±0.07)
Wild store	1.78 (±0.07)	2.38 (±0.14)
Atomicity violation	1.28 (±0.02)	1.40 (±0.08)
Memory leak	1.06 (±0.01)	1.40 (±0.14)
Lock contention	1.17 (±0.01)	2.16 (±0.24)
Apache 45605	1.07 (±0.03)	1.27 (±0.03)
Apache 25520	1.01 (±0.00)	1.17 (±0.00)

Table 2: Skew. The reported values are the longest epoch execution time divided by the average execution time.

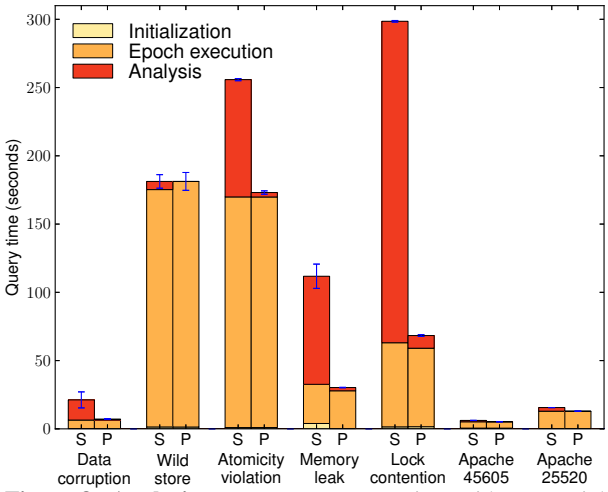


Figure 8: Analysis. We compare query time with sequential (S) and parallel (P) analysis using 64 cores. The regions in each bar show how much time is spent in each phase along the critical path of query processing.

6.4 Benefit of parallel analysis

We next quantify how much benefit is achieved by parallelizing analysis. Figure 8 compares query response time for sequential analysis and parallel analysis using the analyzers for each query described in Section 4. We show results with 64 cores, i.e., the largest number of cores we can support without emulation.

All scenarios except the wild store and Apache scenarios achieve substantial speedup by parallelizing analysis. The atomicity violation, lock contention, and memory leak analyses traverse large tracelogs and track complex interactions across log messages. Many of these interactions are contained within a single epoch, so local analysis can resolve them. Using parallel analysis speeds up analysis by up to a factor of 96, with a mean improvement of 31. Overall, parallel analysis accelerates total query time by up to a factor of 4, with a mean improvement of 2. Sequential analysis does not scale, so we expect this speedup to increase as the cluster size grows.

Benchmark	Fork-based query time(s)	Compiler-based query time(s)	Speedup
Data corruption	7.5 (±0.9)	.79 (±0.1)	9.6 (±1.4)
Memory leak	32.5 (±0.2)	2.30 (±0.1)	14.2 (±0.5)
Lock contention	632.2 (±5.0)	6.01 (±0.0)	105.3 (±1.0)

Table 3: Isolation performance. We compare the time to execute the first 64 out of 1024 epochs using fork-based and compiler-based isolation.

6.5 Isolation

Table 3 compares the performance of compiler-based and fork-based isolation for all queries that do not use continuous function evaluation (which requires compiler-based isolation). On average, compiler-based isolation speeds up epoch execution by a factor of 24, making it the best choice unless its restrictions on what can be included in a tracer become too onerous.

6.6 Recording Overhead

We measured recording overhead on a server with an 8-core Xeon E5620 2.4 GHz processor, 6 GB memory, and two 1 TB 7200 RPM hard drives. The average recording overhead for our application benchmarks was 6%. Checkpointing every two seconds increases the average overhead to 8%, and adding additional logging for retro-timing increases average overhead to 9%. The additional space overhead for retro-timing is 17% compared to the base Arnold logging.

7 Related Work

Sledgehammer is the first general-purpose framework for accelerating debugging tools by parallelizing them across a cluster. It has frequently been observed that deterministic replay [11] is a great help in debugging [5, 17, 27, 32, 36]. Sledgehammer leverages Arnold [10] replay both to ensure that results of successive queries are consistent and also to parallelize work via epoch parallelism [35]. JetStream [29] uses epoch parallelism for a different task: dynamic information flow tracking (DIFT). Sledgehammer's tracer isolation has less overhead and scales much better than the dynamic binary instrumentation used by JetStream, making it better suited for tasks like debugging that need not monitor every instruction executed.

Many tools aim to simplify and optimize the dynamic tracing of program execution. Dtrace and SystemTap reduce overhead when tracing is not being used, but are expensive when gathering large traces [4, 28]. Execution mining [19] treats executions as data streams that can be dynamically analyzed and supports iterative queries by indexing and caching streams. Other tools introspect distributed systems. Fay [12] lets users introspect at the start and end of functions but injected code must be side-effect free. Pivot tracing [23] lets users specify queries in an SQL-like language. These tools help debug parallel programs, but, unlike Sledgehammer, they are not themselves parallelized for performance.

Several prior systems support retro-logging. Most isolate all code added to an execution using fork-based approaches [6, 15]; this comes with high overhead. Others use binary rewriting approaches for isolation such as Pin and Valgrind; these tools do not scale to thousands of cores [29]. Sledgehammer reduces isolation overhead through compiler-based isolation and hides remaining overhead through parallelization. Like Sledgehammer, rdb [14] allows users to modify source code and executes the modifications during replay; however, rdb prohibits program state modifications instead of isolating them. Dora [36] allows the added code to perturb application state and uses mutable replay to make a best effort to keep replaying the application correctly after the perturbation. This eliminates isolation overhead, but there is no guarantee that the debugging output will be correct. Mutable replay is a good choice when output is simple and can be verified by inspection, but incorrect results could prove frustrating for complex debugging tasks.

As documented in the wild store scenario, developers commonly write debug functions to verify invariants. Researchers have advocated running similar functions at strategic code locations to repair structures [9] or detect likely invariants [13]. Continuous function evaluation takes this to an extreme by logically running a function after every instruction. X-Ray [1] systematically measures timing during recording to support profiling of replayed executions; Sledgehammer's more general interface allows debuggers to define the events being measured and understand the uncertainty in timing results.

8 Conclusion

Sledgehammer is a cluster-fueled debugger: it makes powerful debugging tools interactive by parallelizing application and tool execution, as well as analysis, across many cores in a cluster. This makes tools such as parallel retro-logging, continuous function evaluation, and retro-timing practical by running them an average of 416 times faster than sequential execution on a 1024-core cluster.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their thoughtful comments. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441, and by NSF GRFP and MSR Ph.D Fellowships. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[2] Bug 25520. `https://bz.apache.org/bugzilla/show_bug.cgi?id=25520`.

[3] Bug 45605. `https://bz.apache.org/bugzilla/show_bug.cgi?id=45605`.

[4] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 15–28.

[5] CHEN, P., AND NOBLE, B. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems)* (Schloss Elmau, Germany, May 2001).

[6] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 1–14.

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[8] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (February 2013), 74–80.

[9] DEMSKY, B., ERNST, M. D., GUO, P. J., MCCARMANT, S., PERKINS, J. H., AND RINARD, M. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analaysis* (July 2006).

[10] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[12] ERLINGSSON, U., PEINADO, M., PETER, S., AND BUDIU, M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (October 2011), pp. 311–326.

[13] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (February 2001).

[14] HONARMAND, N., AND TORRELLAS, J. Replay debugging: Leveraging record and replay for program debugging. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (June 2014), pp. 455–456.

[15] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[16] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[17] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference* (April 2005), pp. 1–15.

[18] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization* (2004).

[19] LEFEBVRE, G., CULLY, B., HEAD, C., SPEAR, M., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Execution Mining. In *Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2012).

[20] LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y., AND ZHAI, C. Have things changed now?: an empirical study of bug characteristics in modern

open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (2006), ACM, pp. 25–33.

[21] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 329–339.

[22] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[23] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (2015).

[24] MCCONNELL, S. *Code complete*. Pearson Education, 2004.

[25] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached, June 2013. `https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/`.

[26] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007).

[27] O'CALLAHAN, R., JONES, C., FROYD, N., HUEY, K., NOLL, A., AND PARTUSH, N. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, July 2017).

[28] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium* (Ottawa, ON, Canada, July 2005), pp. 49–64.

[29] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation* (Savannah, GA, November 2016).

[30] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[31] rr: lightweight recording and deterministic debugging. `http://www.rr-project.org`.

[32] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[33] TALLENT, N. R., MELLOR-CRUMMEY, J. M., AND PORTERFIELD, A. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 269–280.

[34] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).

[35] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[36] VIENNOT, N., NAIR, S., AND NIEH, J. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2013).

[37] YU, J., AND NARAYANASAMY, S. A case for an interleaving constrained shared-memory multiprocessor. In *Proceedings of the 36th International Symposium on Computer Architecture* (June 2009), pp. 325–336.

[38] YUAN, D., PARK, S., AND ZHOU, Y. Characterising logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, June 2012).

Ray: A Distributed Framework for Emerging AI Applications

Philipp Moritz*, Robert Nishihara*, Stephanie Wang, Alexey Tumanov, Richard Liaw,
Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, Ion Stoica
University of California, Berkeley

Abstract

The next generation of AI applications will continuously interact with the environment and learn from these interactions. These applications impose new and demanding systems requirements, both in terms of performance and flexibility. In this paper, we consider these requirements and present Ray—a distributed system to address them. Ray implements a unified interface that can express both task-parallel and actor-based computations, supported by a single dynamic execution engine. To meet the performance requirements, Ray employs a distributed scheduler and a distributed and fault-tolerant store to manage the system's control state. In our experiments, we demonstrate scaling beyond 1.8 million tasks per second and better performance than existing specialized systems for several challenging reinforcement learning applications.

1 Introduction

Over the past two decades, many organizations have been collecting—and aiming to exploit—ever-growing quantities of data. This has led to the development of a plethora of frameworks for distributed data analysis, including batch [20, 64, 28], streaming [15, 39, 31], and graph [34, 35, 24] processing systems. The success of these frameworks has made it possible for organizations to analyze large data sets as a core part of their business or scientific strategy, and has ushered in the age of "Big Data."

More recently, the scope of data-focused applications has expanded to encompass more complex artificial intelligence (AI) or machine learning (ML) techniques [30]. The paradigm case is that of *supervised learning*, where data points are accompanied by labels, and where the workhorse technology for mapping data points to labels is provided by deep neural networks. The complexity of these deep networks has led to another flurry of frameworks that focus on the training of deep neural networks and their use in prediction. These frameworks often leverage specialized hardware (e.g., GPUs and TPUs), with the goal of reducing training time in a batch setting. Examples include TensorFlow [7], MXNet [18], and PyTorch [46].

The promise of AI is, however, far broader than classical supervised learning. Emerging AI applications must increasingly operate in dynamic environments, react to changes in the environment, and take sequences of actions to accomplish long-term goals [8, 43]. They must aim not only to exploit the data gathered, but also to explore the space of possible actions. These broader requirements are naturally framed within the paradigm of *reinforcement learning* (RL). RL deals with learning to operate continuously within an uncertain environment based on delayed and limited feedback [56]. RL-based systems have already yielded remarkable results, such as Google's AlphaGo beating a human world champion [54], and are beginning to find their way into dialogue systems, UAVs [42], and robotic manipulation [25, 60].

The central goal of an RL application is to learn a policy—a mapping from the state of the environment to a choice of action—that yields effective performance over time, e.g., winning a game or piloting a drone. Finding effective policies in large-scale applications requires three main capabilities. First, RL methods often rely on *simulation* to evaluate policies. Simulations make it possible to explore many different choices of action sequences and to learn about the long-term consequences of those choices. Second, like their supervised learning counterparts, RL algorithms need to perform *distributed training* to improve the policy based on data generated through simulations or interactions with the physical environment. Third, policies are intended to provide solutions to control problems, and thus it is necessary to *serve* the policy in interactive closed-loop and open-loop control scenarios.

These characteristics drive new systems requirements: a system for RL must support *fine-grained* computations (e.g., rendering actions in milliseconds when interacting with the real world, and performing vast numbers of sim-

*equal contribution

ulations), must support *heterogeneity* both in time (e.g., a simulation may take milliseconds or hours) and in resource usage (e.g., GPUs for training and CPUs for simulations), and must support *dynamic* execution, as results of simulations or interactions with the environment can change future computations. Thus, we need a dynamic computation framework that handles millions of heterogeneous tasks per second at millisecond-level latencies.

Existing frameworks that have been developed for Big Data workloads or for supervised learning workloads fall short of satisfying these new requirements for RL. Bulk-synchronous parallel systems such as MapReduce [20], Apache Spark [64], and Dryad [28] do not support fine-grained simulation or policy serving. Task-parallel systems such as CIEL [40] and Dask [48] provide little support for distributed training and serving. The same is true for streaming systems such as Naiad [39] and Storm [31]. Distributed deep-learning frameworks such as TensorFlow [7] and MXNet [18] do not naturally support simulation and serving. Finally, model-serving systems such as TensorFlow Serving [6] and Clipper [19] support neither training nor simulation.

While in principle one could develop an end-to-end solution by stitching together several existing systems (e.g., Horovod [53] for distributed training, Clipper [19] for serving, and CIEL [40] for simulation), in practice this approach is untenable due to the *tight coupling* of these components within applications. As a result, researchers and practitioners today build one-off systems for specialized RL applications [58, 41, 54, 44, 49, 5]. This approach imposes a massive systems engineering burden on the development of distributed applications by essentially pushing standard systems challenges like scheduling, fault tolerance, and data movement onto each application.

In this paper, we propose Ray, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications. The requirements of these workloads range from lightweight and stateless computations, such as for simulation, to long-running and stateful computations, such as for training. To satisfy these requirements, Ray implements a unified interface that can express both *task-parallel* and *actor-based* computations. *Tasks* enable Ray to efficiently and dynamically load balance simulations, process large inputs and state spaces (e.g., images, video), and recover from failures. In contrast, *actors* enable Ray to efficiently support stateful computations, such as model training, and expose shared mutable state to clients, (e.g., a parameter server). Ray implements the actor and the task abstractions on top of a single dynamic execution engine that is highly scalable and fault tolerant.

To meet the performance requirements, Ray distributes two components that are typically centralized in existing frameworks [64, 28, 40]: (1) the task scheduler and (2) a

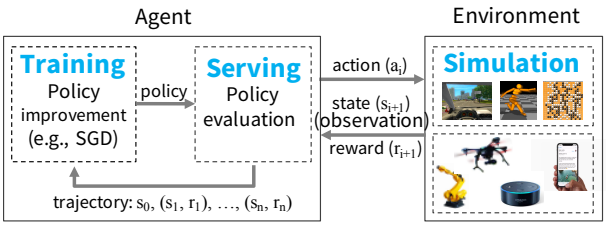


Figure 1: Example of an RL system.

metadata store which maintains the computation lineage and a directory for data objects. This allows Ray to schedule millions of tasks per second with millisecond-level latencies. Furthermore, Ray provides lineage-based fault tolerance for tasks and actors, and replication-based fault tolerance for the metadata store.

While Ray supports serving, training, and simulation in the context of RL applications, this does not mean that it should be viewed as a replacement for systems that provide solutions for these workloads in other contexts. In particular, Ray does not aim to substitute for serving systems like Clipper [19] and TensorFlow Serving [6], as these systems address a broader set of challenges in deploying models, including model management, testing, and model composition. Similarly, despite its flexibility, Ray is not a substitute for generic data-parallel frameworks, such as Spark [64], as it currently lacks the rich functionality and APIs (e.g., straggler mitigation, query optimization) that these frameworks provide.

We make the following **contributions**:

- We design and build the first distributed framework that unifies training, simulation, and serving—necessary components of emerging RL applications.
- To support these workloads, we unify the actor and task-parallel abstractions on top of a dynamic task execution engine.
- To achieve scalability and fault tolerance, we propose a system design principle in which control state is stored in a sharded metadata store and all other system components are stateless.
- To achieve scalability, we propose a bottom-up distributed scheduling strategy.

2 Motivation and Requirements

We begin by considering the basic components of an RL system and fleshing out the key requirements for Ray. As shown in Figure 1, in an RL setting, an *agent* interacts repeatedly with the *environment*. The goal of the agent is to learn a policy that maximizes a *reward*. A *policy* is

```
// evaluate policy by interacting with env. (e.g., simulator)
rollout(policy, environment):
  trajectory = []
  state = environment.initial_state()
  while (not environment.has_terminated()):
    action = policy.compute(state) // Serving
    state, reward = environment.step(action) // Simulation
    trajectory.append(state, reward)
  return trajectory

// improve policy iteratively until it converges
train_policy(environment):
  policy = initial_policy()
  while (policy has not converged):
    trajectories = []
    for i from 1 to k:
      // evaluate policy by generating k rollouts
      trajectories.append(rollout(policy, environment))
      // improve policy
    policy = policy.update(trajectories) // Training
  return policy
```

Figure 2: Typical RL pseudocode for learning a policy.

a mapping from the state of the environment to a choice of *action*. The precise definitions of environment, agent, state, action, and reward are application-specific.

To learn a policy, an agent typically employs a two-step process: (1) *policy evaluation* and (2) *policy improvement*. To evaluate the policy, the agent interacts with the environment (e.g., with a simulation of the environment) to generate *trajectories*, where a trajectory consists of a sequence of (state, reward) tuples produced by the current policy. Then, the agent uses these trajectories to improve the policy; i.e., to update the policy in the direction of the gradient that maximizes the reward. Figure 2 shows an example of the pseudocode used by an agent to learn a policy. This pseudocode evaluates the policy by invoking **rollout**(*environment*, *policy*) to generate trajectories. **train_policy**() then uses these trajectories to improve the current policy via *policy*.**update**(*trajectories*). This process repeats until the policy converges.

Thus, a framework for RL applications must provide efficient support for *training*, *serving*, and *simulation* (Figure 1). Next, we briefly describe these workloads.

Training typically involves running stochastic gradient descent (SGD), often in a distributed setting, to update the policy. Distributed SGD typically relies on an allreduce aggregation step or a parameter server [32].

Serving uses the trained policy to render an action based on the current state of the environment. A serving system aims to minimize latency, and maximize the number of decisions per second. To scale, load is typically balanced across multiple nodes serving the policy.

Finally, most existing RL applications use *simulations* to evaluate the policy—current RL algorithms are not sample-efficient enough to rely solely on data obtained from interactions with the physical world. These simulations vary widely in complexity. They might take a few ms (e.g., simulate a move in a chess game) to minutes (e.g., simulate a realistic environment for a self-driving car).

In contrast with supervised learning, in which training and serving can be handled separately by different systems, in RL *all three of these workloads are tightly coupled in a single application*, with stringent latency requirements between them. Currently, no framework supports this coupling of workloads. In theory, multiple specialized frameworks could be stitched together to provide the overall capabilities, but in practice, the resulting data movement and latency between systems is prohibitive in the context of RL. As a result, researchers and practitioners have been building their own one-off systems.

This state of affairs calls for the development of new distributed frameworks for RL that can efficiently support training, serving, and simulation. In particular, such a framework should satisfy the following requirements:

Fine-grained, heterogeneous computations. The duration of a computation can range from milliseconds (e.g., taking an action) to hours (e.g., training a complex policy). Additionally, training often requires heterogeneous hardware (e.g., CPUs, GPUs, or TPUs).

Flexible computation model. RL applications require both stateless and stateful computations. Stateless computations can be executed on any node in the system, which makes it easy to achieve load balancing and movement of computation to data, if needed. Thus stateless computations are a good fit for fine-grained simulation and data processing, such as extracting features from images or videos. In contrast stateful computations are a good fit for implementing parameter servers, performing repeated computation on GPU-backed data, or running third-party simulators that do not expose their state.

Dynamic execution. Several components of RL applications require dynamic execution, as the order in which computations finish is not always known in advance (e.g., the order in which simulations finish), and the results of a computation can determine future computations (e.g., the results of a simulation will determine whether we need to perform more simulations).

We make two final comments. First, to achieve high utilization in large clusters, such a framework must handle *millions of tasks per second*.* Second, such a framework is not intended for implementing deep neural networks or complex simulators from scratch. Instead, it should enable seamless integration with existing simulators [13, 11, 59] and deep learning frameworks [7, 18, 46, 29].

*Assume 5ms single-core tasks and a cluster of 200 32-core nodes. This cluster can run $(1s/5ms) \times 32 \times 200 = 1.28$M tasks/sec.

Name	Description
$futures = \textbf{f.remote}(args)$	Execute function f remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
$objects = \textbf{ray.get}(futures)$	Return the values associated with one or more futures. This is blocking.
$ready_futures = \textbf{ray.wait}(futures, k, timeout)$	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
$actor = \textbf{Class.remote}(args)$ $futures = actor.\textbf{method.remote}(args)$	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

Table 1: Ray API

3 Programming and Computation Model

Ray implements a dynamic task graph computation model, i.e., it models an application as a graph of dependent tasks that evolves during execution. On top of this model, Ray provides both an actor and a task-parallel programming abstraction. This unification differentiates Ray from related systems like CIEL, which only provides a task-parallel abstraction, and from Orleans [14] or Akka [1], which primarily provide an actor abstraction.

3.1 Programming Model

Tasks. A *task* represents the execution of a remote function on a stateless worker. When a remote function is invoked, a *future* representing the result of the task is returned immediately. Futures can be retrieved using **ray.get**() and passed as arguments into other remote functions without waiting for their result. This allows the user to express parallelism while capturing data dependencies. Table 1 shows Ray's API.

Remote functions operate on immutable objects and are expected to be *stateless* and side-effect free: their outputs are determined solely by their inputs. This implies idempotence, which simplifies fault tolerance through function re-execution on failure.

Actors. An *actor* represents a stateful computation. Each actor exposes methods that can be invoked remotely and are executed serially. A method execution is similar to a task, in that it executes remotely and returns a future, but differs in that it executes on a *stateful* worker. A *handle* to an actor can be passed to other actors or tasks, making it possible for them to invoke methods on that actor.

Tasks (stateless)	Actors (stateful)
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

Table 2: Tasks vs. actors tradeoffs.

Table 2 summarizes the properties of tasks and actors. Tasks enable fine-grained load balancing through leveraging load-aware scheduling at task granularity, input data locality, as each task can be scheduled on the node storing its inputs, and low recovery overhead, as there is no need to checkpoint and recover intermediate state. In contrast, actors provide much more efficient fine-grained updates, as these updates are performed on internal rather than external state, which typically requires serialization and deserialization. For example, actors can be used to implement parameter servers [32] and GPU-based iterative computations (e.g., training). In addition, actors can be used to wrap third-party simulators and other opaque handles that are hard to serialize.

To satisfy the requirements for heterogeneity and flexibility (Section 2), we augment the API in three ways. First, to handle concurrent tasks with heterogeneous durations, we introduce **ray.wait**(), which waits for the first k available results, instead of waiting for *all* results like **ray.get**(). Second, to handle resource-heterogeneous tasks, we enable developers to specify resource requirements so that the Ray scheduler can efficiently manage resources. Third, to improve flexibility, we enable *nested remote functions*, meaning that remote functions can invoke other remote functions. This is also critical for achieving high scalability (Section 4), as it enables multiple processes to invoke remote functions in a distributed fashion.

3.2 Computation Model

Ray employs a dynamic task graph computation model [21], in which the execution of both remote functions and actor methods is automatically triggered by the system when their inputs become available. In this section, we describe how the computation graph (Figure 4) is constructed from a user program (Figure 3). This program uses the API in Table 1 to implement the pseudocode from Figure 2.

Ignoring actors first, there are two types of nodes in a computation graph: data objects and remote function invocations, or tasks. There are also two types of edges: data edges and control edges. Data edges capture the de-

```python
@ray.remote
def create_policy():
  # Initialize the policy randomly.
  return policy

@ray.remote(num_gpus=1)
class Simulator(object):
  def __init__(self):
    # Initialize the environment.
    self.env = Environment()
  def rollout(self, policy, num_steps):
    observations = []
    observation = self.env.current_state()
    for _ in range(num_steps):
      action = policy(observation)
      observation = self.env.step(action)
      observations.append(observation)
    return observations

@ray.remote(num_gpus=2)
def update_policy(policy, *rollouts):
  # Update the policy.
  return policy

@ray.remote
def train_policy():
  # Create a policy.
  policy_id = create_policy.remote()
  # Create 10 actors.
  simulators = [Simulator.remote() for _ in range(10)]
  # Do 100 steps of training.
  for _ in range(100):
    # Perform one rollout on each actor.
    rollout_ids = [s.rollout.remote(policy_id)
                   for s in simulators]
    # Update the policy with the rollouts.
    policy_id =
        update_policy.remote(policy_id, *rollout_ids)
  return ray.get(policy_id)
```

Figure 3: Python code implementing the example in Figure 2 in Ray. Note that @ray.remote indicates remote functions and actors. Invocations of remote functions and actor methods return futures, which can be passed to subsequent remote functions or actor methods to encode task dependencies. Each actor has an environment object self.env shared between all of its methods.

pendencies between data objects and tasks. More precisely, if data object D is an output of task T, we add a data edge from T to D. Similarly, if D is an input to T, we add a data edge from D to T. Control edges capture the computation dependencies that result from nested remote functions (Section 3.1): if task T_1 invokes task T_2, then we add a control edge from T_1 to T_2.

Actor method invocations are also represented as nodes in the computation graph. They are identical to tasks with one key difference. To capture the state dependency across subsequent method invocations on the same actor, we add a third type of edge: a stateful edge. If method M_j is called right after method M_i on the same actor, then we add a stateful edge from M_i to M_j. Thus, all

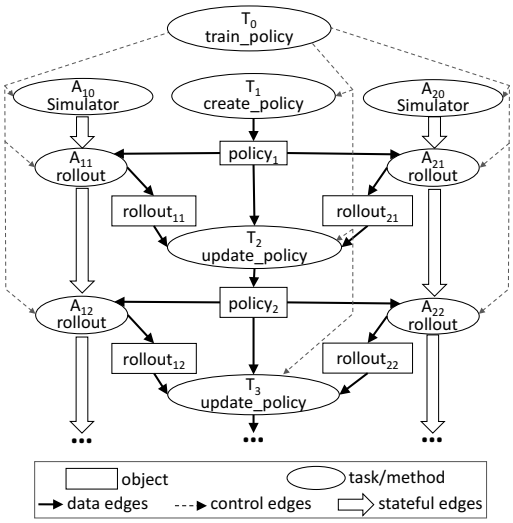


Figure 4: The task graph corresponding to an invocation of train_policy.remote() in Figure 3. Remote function calls and the actor method calls correspond to tasks in the task graph. The figure shows two actors. The method invocations for each actor (the tasks labeled A_{1i} and A_{2i}) have stateful edges between them indicating that they share the mutable actor state. There are control edges from train_policy to the tasks that it invokes. To train multiple policies in parallel, we could call train_policy.remote() multiple times.

methods invoked on the same actor object form a chain that is connected by stateful edges (Figure 4). This chain captures the order in which these methods were invoked.

Stateful edges help us embed actors in an otherwise stateless task graph, as they capture the implicit data dependency between successive method invocations sharing the internal state of an actor. Stateful edges also enable us to maintain lineage. As in other dataflow systems [64], we track data lineage to enable reconstruction. By explicitly including stateful edges in the lineage graph, we can easily reconstruct lost data, whether produced by remote functions or actor methods (Section 4.2.3).

4 Architecture

Ray's architecture comprises (1) an application layer implementing the API, and (2) a system layer providing high scalability and fault tolerance.

4.1 Application Layer

The application layer consists of three types of processes:

- *Driver*: A process executing the user program.
- *Worker*: A stateless process that executes tasks (remote functions) invoked by a driver or another

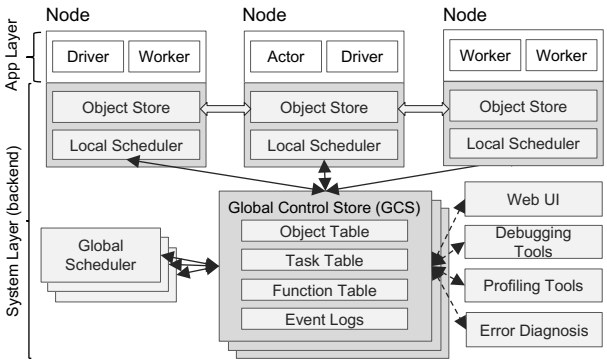


Figure 5: Ray's architecture consists of two parts: an *application* layer and a *system* layer. The application layer implements the API and the computation model described in Section 3, the system layer implements task scheduling and data management to satisfy the performance and fault-tolerance requirements.

worker. Workers are started automatically and assigned tasks by the system layer. When a remote function is declared, the function is automatically published to all workers. A worker executes tasks serially, with no local state maintained across tasks.

- *Actor*: A stateful process that executes, when invoked, only the methods it exposes. Unlike a worker, an actor is explicitly instantiated by a worker or a driver. Like workers, actors execute methods serially, except that each method depends on the state resulting from the previous method execution.

4.2 System Layer

The system layer consists of three major components: a global control store, a distributed scheduler, and a distributed object store. All components are horizontally scalable and fault-tolerant.

4.2.1 Global Control Store (GCS)

The global control store (GCS) maintains the entire control state of the system, and it is a unique feature of our design. At its core, GCS is a key-value store with pub-sub functionality. We use sharding to achieve scale, and per-shard chain replication [61] to provide fault tolerance. The primary reason for the GCS and its design is to maintain fault tolerance and low latency for a system that can dynamically spawn millions of tasks per second.

Fault tolerance in case of node failure requires a solution to maintain lineage information. Existing lineage-based solutions [64, 63, 40, 28] focus on coarse-grained parallelism and can therefore use a single node (e.g., master, driver) to store the lineage without impacting performance. However, this design is not scalable for a fine-grained and dynamic workload like simulation. Therefore, we decouple the durable lineage storage from the other system components, allowing each to scale independently.

Maintaining low latency requires minimizing overheads in task scheduling, which involves choosing where to execute, and subsequently task dispatch, which involves retrieving remote inputs from other nodes. Many existing dataflow systems [64, 40, 48] couple these by storing object locations and sizes in a centralized scheduler, a natural design when the scheduler is not a bottleneck. However, the scale and granularity that Ray targets requires keeping the centralized scheduler off the critical path. Involving the scheduler in each object transfer is prohibitively expensive for primitives important to distributed training like allreduce, which is both communication-intensive and latency-sensitive. Therefore, we store the object metadata in the GCS rather than in the scheduler, fully decoupling task dispatch from task scheduling.

In summary, the GCS significantly simplifies Ray's overall design, as it *enables every component in the system to be stateless*. This not only simplifies support for fault tolerance (i.e., on failure, components simply restart and read the lineage from the GCS), but also makes it easy to scale the distributed object store and scheduler independently, as all components share the needed state via the GCS. An added benefit is the easy development of debugging, profiling, and visualization tools.

4.2.2 Bottom-Up Distributed Scheduler

As discussed in Section 2, Ray needs to dynamically schedule millions of tasks per second, tasks which may take as little as a few milliseconds. None of the cluster schedulers we are aware of meet these requirements. Most cluster computing frameworks, such as Spark [64], CIEL [40], and Dryad [28] implement a centralized scheduler, which can provide locality but at latencies in the tens of ms. Distributed schedulers such as work stealing [12], Sparrow [45] and Canary [47] can achieve high scale, but they either don't consider data locality [12], or assume tasks belong to independent jobs [45], or assume the computation graph is known [47].

To satisfy the above requirements, we design a two-level hierarchical scheduler consisting of a global scheduler and per-node local schedulers. To avoid overloading the global scheduler, the tasks created at a node are submitted first to the node's local scheduler. A local scheduler schedules tasks locally unless the node is overloaded (i.e., its local task queue exceeds a predefined threshold), or it cannot satisfy a task's requirements (e.g., lacks a GPU). If a local scheduler decides not to schedule a task locally, it forwards it to the global scheduler. Since this scheduler attempts to schedule tasks locally first (i.e., at the leaves of the scheduling hierarchy), we call it a *bottom-up scheduler*.

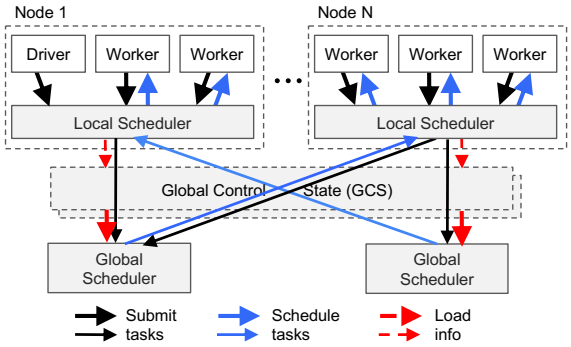


Figure 6: Bottom-up distributed scheduler. Tasks are submitted bottom-up, from drivers and workers to a local scheduler and forwarded to the global scheduler only if needed (Section 4.2.2). The thickness of each arrow is proportional to its request rate.

The global scheduler considers each node's load and task's constraints to make scheduling decisions. More precisely, the global scheduler identifies the set of nodes that have enough resources of the type requested by the task, and of these nodes selects the node which provides the lowest *estimated waiting time*. At a given node, this time is the sum of (i) the estimated time the task will be queued at that node (i.e., task queue size times average task execution), and (ii) the estimated transfer time of task's remote inputs (i.e., total size of remote inputs divided by average bandwidth). The global scheduler gets the queue size at each node and the node resource availability via heartbeats, and the location of the task's inputs and their sizes from GCS. Furthermore, the global scheduler computes the average task execution and the average transfer bandwidth using simple exponential averaging. If the global scheduler becomes a bottleneck, we can instantiate more replicas all sharing the same information via GCS. This makes our scheduler architecture highly scalable.

4.2.3 In-Memory Distributed Object Store

To minimize task latency, we implement an in-memory distributed storage system to store the inputs and outputs of every task, or stateless computation. On each node, we implement the object store via *shared memory*. This allows zero-copy data sharing between tasks running on the same node. As a data format, we use Apache Arrow [2].

If a task's inputs are not local, the inputs are replicated to the local object store before execution. Also, a task writes its outputs to the local object store. Replication eliminates the potential bottleneck due to hot data objects and minimizes task execution time as a task only reads/writes data from/to the local memory. This increases throughput for computation-bound workloads, a profile shared by many AI applications. For low latency, we keep objects entirely in memory and evict them as needed to disk using an LRU policy.

As with existing cluster computing frameworks, such as Spark [64], and Dryad [28], the object store is limited to *immutable data*. This obviates the need for complex consistency protocols (as objects are not updated), and simplifies support for fault tolerance. In the case of node failure, Ray recovers any needed objects through lineage re-execution. The lineage stored in the GCS tracks both stateless tasks and stateful actors during initial execution; we use the former to reconstruct objects in the store.

For simplicity, our object store does not support distributed objects, i.e., each object fits on a single node. Distributed objects like large matrices or trees can be implemented at the application level as collections of futures.

4.2.4 Implementation

Ray is an active open source project[†] developed at the University of California, Berkeley. Ray fully integrates with the Python environment and is easy to install by simply running `pip install ray`. The implementation comprises $\approx$ 40K lines of code (LoC), 72% in C++ for the system layer, 28% in Python for the application layer. The GCS uses one Redis [50] key-value store per shard, with entirely single-key operations. GCS tables are sharded by object and task IDs to scale, and every shard is chain-replicated [61] for fault tolerance. We implement both the local and global schedulers as event-driven, single-threaded processes. Internally, local schedulers maintain cached state for local object metadata, tasks waiting for inputs, and tasks ready for dispatch to a worker. To transfer large objects between different object stores, we stripe the object across multiple TCP connections.

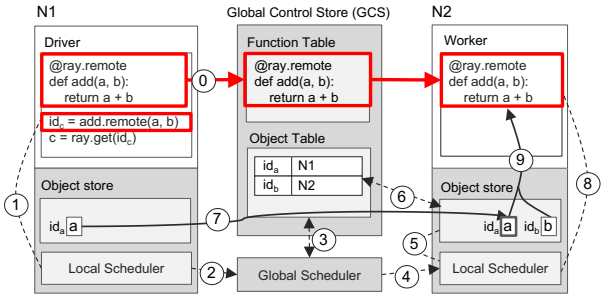
4.3 Putting Everything Together

Figure 7 illustrates how Ray works end-to-end with a simple example that adds two objects a and b, which could be scalars or matrices, and returns result c. The remote function **add()** is automatically registered with the GCS upon initialization and distributed to every worker in the system (step 0 in Figure 7a).

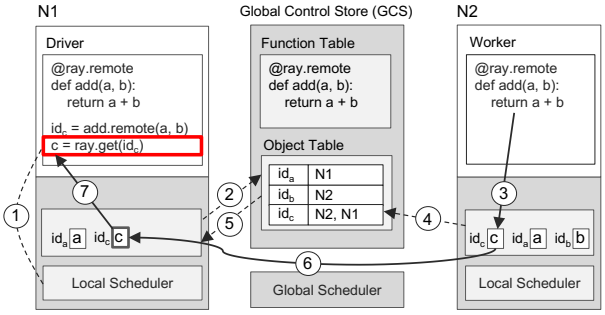
Figure 7a shows the step-by-step operations triggered by a driver invoking **add.remote**(a, b), where a and b are stored on nodes $N1$ and $N2$, respectively. The driver submits **add**(a, b) to the local scheduler (step 1), which forwards it to a global scheduler (step 2).[‡] Next, the global scheduler looks up the locations of **add**(a, b)'s arguments in the GCS (step 3) and decides to schedule the task on node $N2$, which stores argument b (step 4). The local scheduler at node $N2$ checks whether the local object store contains **add**(a, b)'s arguments (step 5). Since the

[†] `https://github.com/ray-project/ray`

[‡] Note that $N1$ could also decide to schedule the task locally.



(a) Executing a task remotely



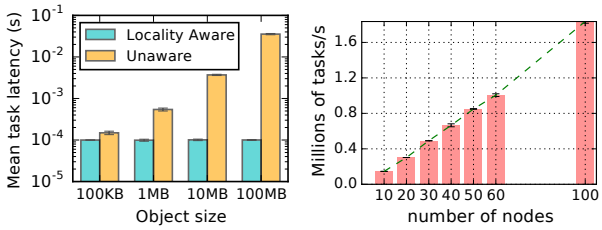
(b) Returning the result of a remote task

Figure 7: An end-to-end example that adds a and b and returns c. Solid lines are data plane operations and dotted lines are control plane operations. (a) The function **add()** is registered with the GCS by node 1 ($N1$), invoked on $N1$, and executed on $N2$. (b) $N1$ gets **add()**'s result using **ray.get()**. The Object Table entry for c is created in step 4 and updated in step 6 after c is copied to $N1$.

local store doesn't have object a, it looks up a's location in the GCS (step 6). Learning that a is stored at $N1$, $N2$'s object store replicates it locally (step 7). As all arguments of **add()** are now stored locally, the local scheduler invokes **add()** at a local worker (step 8), which accesses the arguments via shared memory (step 9).

Figure 7b shows the step-by-step operations triggered by the execution of **ray.get()** at $N1$, and of **add()** at $N2$, respectively. Upon **ray.get**(id_c)'s invocation, the driver checks the local object store for the value c, using the future id_c returned by **add()** (step 1). Since the local object store doesn't store c, it looks up its location in the GCS. At this time, there is no entry for c, as c has not been created yet. As a result, $N1$'s object store registers a callback with the Object Table to be triggered when c's entry has been created (step 2). Meanwhile, at $N2$, **add()** completes its execution, stores the result c in the local object store (step 3), which in turn adds c's entry to the GCS (step 4). As a result, the GCS triggers a callback to $N1$'s object store with c's entry (step 5). Next, $N1$ replicates c from $N2$ (step 6), and returns c to **ray.get()** (step 7), which finally completes the task.

While this example involves a large number of RPCs, in many cases this number is much smaller, as most tasks are scheduled locally, and the GCS replies are cached by the global and local schedulers.



(a) Ray locality scheduling (b) Ray scalability

Figure 8: (a) Tasks leverage locality-aware placement. 1000 tasks with a random object dependency are scheduled onto one of two nodes. With locality-aware policy, task latency remains independent of the size of task inputs instead of growing by 1-2 orders of magnitude. (b) Near-linear scalability leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 nodes. $x \in \{70, 80, 90\}$ omitted due to cost.

5 Evaluation

In our evaluation, we study the following questions:

1. How well does Ray meet the latency, scalability, and fault tolerance requirements listed in Section 2? (Section 5.1)
2. What overheads are imposed on distributed primitives (e.g., allreduce) written using Ray's API? (Section 5.1)
3. In the context of RL workloads, how does Ray compare against specialized systems for training, serving, and simulation? (Section 5.2)
4. What advantages does Ray provide for RL applications, compared to custom systems? (Section 5.3)

All experiments were run on Amazon Web Services. Unless otherwise stated, we use m4.16xlarge CPU instances and p3.16xlarge GPU instances.

5.1 Microbenchmarks

Locality-aware task placement. Fine-grain load balancing and locality-aware placement are primary benefits of tasks in Ray. Actors, once placed, are unable to move their computation to large remote objects, while tasks can. In Figure 8a, tasks placed without data locality awareness (as is the case for actor methods), suffer 1-2 orders of magnitude latency increase at 10-100MB input data sizes. Ray unifies tasks and actors through the shared object store, allowing developers to use tasks for e.g., expensive postprocessing on output produced by simulation actors.

End-to-end scalability. One of the key benefits of

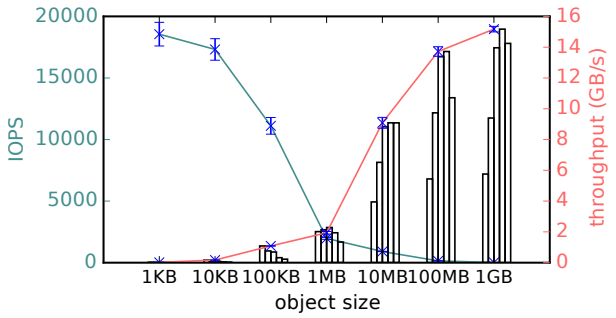
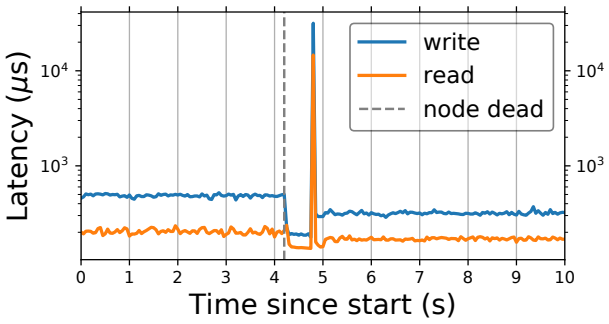
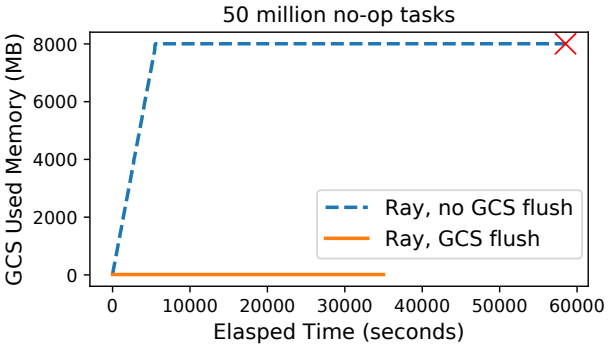


Figure 9: Object store write throughput and IOPS. From a single client, throughput exceeds 15GB/s (red) for large objects and 18K IOPS (cyan) for small objects on a 16 core instance (m4.4xlarge). It uses 8 threads to copy objects larger than 0.5MB and 1 thread for small objects. Bar plots report throughput with 1, 2, 4, 8, 16 threads. Results are averaged over 5 runs.



(a) A timeline for GCS read and write latencies as viewed from a client submitting tasks. The chain starts with 2 replicas. We manually trigger reconfiguration as follows. At $t \approx 4.2$s, a chain member is killed; immediately after, a new chain member joins, initiates state transfer, and restores the chain to 2-way replication. The maximum client-observed latency is under 30ms despite reconfigurations.



(b) The Ray GCS maintains a constant memory footprint with GCS flushing. Without GCS flushing, the memory footprint reaches a maximum capacity and the workload fails to complete within a predetermined duration (indicated by the red cross).

Figure 10: Ray GCS fault tolerance and flushing.

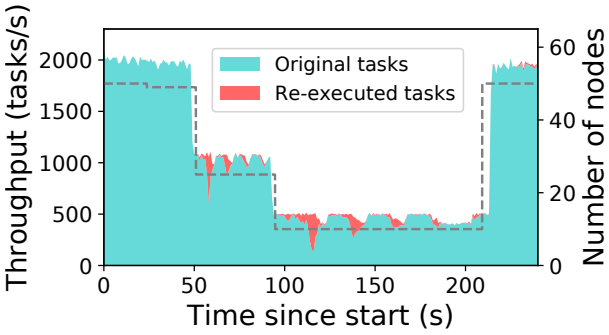
the Global Control Store (GCS) and the bottom-up distributed scheduler is the ability to horizontally scale the system to support a high throughput of fine-grained tasks, while maintaining fault tolerance and low-latency task scheduling. In Figure 8b, we evaluate this ability on an embarrassingly parallel workload of empty tasks, increasing the cluster size on the x-axis. We observe near-perfect linearity in progressively increasing task throughput. Ray exceeds 1 million tasks per second throughput at 60 nodes and continues to scale linearly beyond 1.8 million tasks per second at 100 nodes. The rightmost datapoint shows that Ray can process 100 million tasks in less than a minute (54s), with minimum variability. As expected, increasing task duration reduces throughput proportionally to mean task duration, but the overall scalability remains linear. While many realistic workloads may exhibit more limited scalability due to object dependencies and inherent limits to application parallelism, this demonstrates the scalability of our overall architecture under high load.

Object store performance. To evaluate the performance of the object store (Section 4.2.3), we track two metrics: IOPS (for small objects) and write throughput (for large objects). In Figure 9, the write throughput from a single client exceeds 15GB/s as object size increases. For larger objects, memcpy dominates object creation time. For smaller objects, the main overheads are in serialization and IPC between the client and object store.

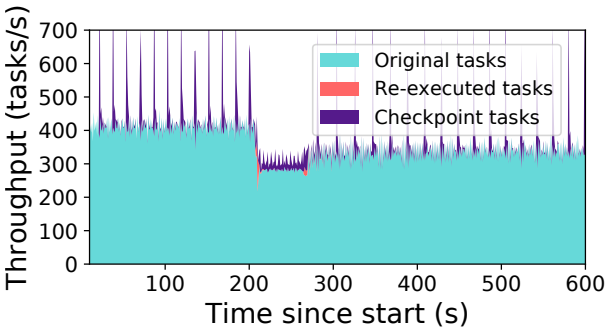
GCS fault tolerance. To maintain low latency while providing strong consistency and fault tolerance, we build a lightweight chain replication [61] layer on top of Redis. Figure 10a simulates recording Ray tasks to and reading tasks from the GCS, where keys are 25 bytes and values are 512 bytes. The client sends requests as fast as it can, having at most one in-flight request at a time. Failures are reported to the chain master either from the client (having received explicit errors, or timeouts despite retries) or from any server in the chain (having received explicit errors). Overall, reconfigurations caused a maximum *client-observed* delay of under 30ms (this includes both failure detection and recovery delays).

GCS flushing. Ray is equipped to periodically flush the contents of GCS to disk. In Figure 10b we submit 50 million empty tasks sequentially and monitor GCS memory consumption. As expected, it grows linearly with the number of tasks tracked and eventually reaches the memory capacity of the system. At that point, the system becomes stalled and the workload fails to finish within a reasonable amount of time. With periodic GCS flushing, we achieve two goals. First, the memory footprint is capped at a user-configurable level (in the microbenchmark we employ an aggressive strategy where consumed memory is kept as low as possible). Second, the flushing mechanism provides a natural way to snapshot lineage to disk for long-running Ray applications.

Recovering from task failures. In Figure 11a, we



(a) Task reconstruction



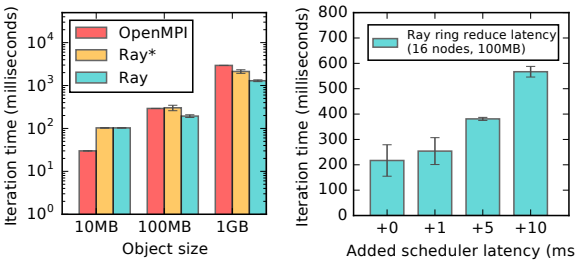
(b) Actor reconstruction

Figure 11: Ray fault-tolerance. **(a)** Ray reconstructs lost task dependencies as nodes are removed (dotted line), and recovers to original throughput when nodes are added back. Each task is 100ms and depends on an object generated by a previously submitted task. **(b)** Actors are reconstructed from their last checkpoint. At $t = 200$s, we kill 2 of the 10 nodes, causing 400 of the 2000 actors in the cluster to be recovered on the remaining nodes ($t = 200$–270s).

demonstrate Ray's ability to transparently recover from worker node failures and elastically scale, using the durable GCS lineage storage. The workload, run on m4.xlarge instances, consists of linear chains of 100ms tasks submitted by the driver. As nodes are removed (at 25s, 50s, 100s), the local schedulers reconstruct previous results in the chain in order to continue execution. Overall *per-node* throughput remains stable throughout.

Recovering from actor failures. By encoding actor method calls as stateful edges directly in the dependency graph, we can reuse the same object reconstruction mechanism as in Figure 11a to provide transparent fault tolerance for *stateful computation*. Ray additionally leverages user-defined checkpoint functions to bound the reconstruction time for actors (Figure 11b). With minimal overhead, checkpointing enables only 500 methods to be re-executed, versus 10k re-executions without checkpointing. In the future, we hope to further reduce actor reconstruction time, e.g., by allowing users to annotate methods that do not mutate state.

Allreduce. Allreduce is a distributed communication



(a) Ray vs OpenMPI (b) Ray scheduler ablation

Figure 12: (a) Mean execution time of allreduce on 16 m4.16xl nodes. Each worker runs on a distinct node. Ray* restricts Ray to 1 thread for sending and 1 thread for receiving. (b) Ray's low-latency scheduling is critical for allreduce.

primitive important to many machine learning workloads. Here, we evaluate whether Ray can natively support a ring allreduce [57] implementation with low enough overhead to match existing implementations [53]. We find that Ray completes allreduce across 16 nodes on 100MB in ∼200ms and 1GB in ∼1200ms, surprisingly outperforming OpenMPI (v1.10), a popular MPI implementation, by 1.5× and 2× respectively (Figure 12a). We attribute Ray's performance to its use of multiple threads for network transfers, taking full advantage of the 25Gbps connection between nodes on AWS, whereas OpenMPI sequentially sends and receives data on a single thread [22]. For smaller objects, OpenMPI outperforms Ray by switching to a lower overhead algorithm, an optimization we plan to implement in the future.

Ray's scheduler performance is critical to implementing primitives such as allreduce. In Figure 12b, we inject artificial task execution delays and show that performance drops nearly 2× with just a few ms of extra latency. Systems with centralized schedulers like Spark and CIEL typically have scheduler overheads in the tens of milliseconds [62, 38], making such workloads impractical. Scheduler *throughput* also becomes a bottleneck since the number of tasks required by ring reduce scales quadratically with the number of participants.

5.2 Building blocks

End-to-end applications (e.g., AlphaGo [54]) require a tight coupling of training, serving, and simulation. In this section, we isolate each of these workloads to a setting that illustrates a typical RL application's requirements. Due to a flexible programming model targeted to RL, and a system designed to support this programming model, Ray matches and sometimes exceeds the performance of dedicated systems for these individual workloads.

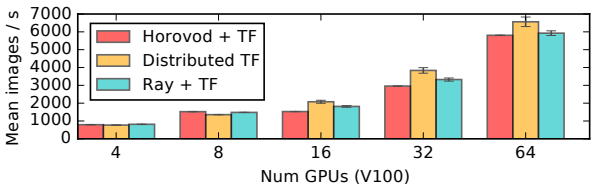


Figure 13: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xl instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in Horovod [53]. We note some measurement deviations from previously reported, likely due to hardware differences and recent TensorFlow performance improvements. We used OpenMPI 3.0, TF 1.8, and NCCL2 for all runs.

5.2.1 Distributed Training

We implement data-parallel synchronous SGD leveraging the Ray actor abstraction to represent model replicas. Model weights are synchronized via allreduce (5.1) or parameter server, both implemented on top of the Ray API.

In Figure 13, we evaluate the performance of the Ray (synchronous) parameter-server SGD implementation against state-of-the-art implementations [53], using the same TensorFlow model and synthetic data generator for each experiment. We compare only against TensorFlow-based systems to accurately measure the overhead imposed by Ray, rather than differences between the deep learning frameworks themselves. In each iteration, model replica actors compute gradients in parallel, send the gradients to a sharded parameter server, then read the summed gradients from the parameter server for the next iteration.

Figure 13 shows that Ray matches the performance of Horovod and is within 10% of distributed TensorFlow (in `distributed_replicated` mode). This is due to the ability to express the same application-level optimizations found in these specialized systems in Ray's general-purpose API. A key optimization is the pipelining of gradient computation, transfer, and summation within a single iteration. To overlap GPU computation with network transfer, we use a custom TensorFlow operator to write tensors directly to Ray's object store.

5.2.2 Serving

Model serving is an important component of end-to-end applications. Ray focuses primarily on the *embedded* serving of models to simulators running within the same dynamic task graph (e.g., within an RL application on Ray). In contrast, systems like Clipper [19] focus on serving predictions to external clients.

In this setting, low latency is critical for achieving high utilization. To show this, in Table 3 we compare the server throughput achieved using a Ray actor to serve a policy versus using the open source Clipper system over REST. Here, both client and server processes are co-located on the same machine (a p3.8xlarge instance). This is often the case for RL applications but not for the general web serving workloads addressed by systems like Clipper. Due to its low-overhead serialization and shared memory abstractions, Ray achieves an order of magnitude higher throughput for a small fully connected policy model that takes in a large input and is also faster on a more expensive residual network policy model, similar to one used in AlphaGo Zero, that takes smaller input.

System	Small Input	Larger Input
Clipper	4400 $\pm$ 15 states/sec	290 $\pm$ 1.3 states/sec
Ray	6200 $\pm$ 21 states/sec	6900 $\pm$ 150 states/sec

Table 3: Throughput comparisons for Clipper [19], a dedicated serving system, and Ray for two embedded serving workloads. We use a residual network and a small fully connected network, taking 10ms and 5ms to evaluate, respectively. The server is queried by clients that each send states of size 4KB and 100KB respectively in batches of 64.

5.2.3 Simulation

Simulators used in RL produce results with variable lengths ("timesteps") that, due to the tight loop with training, must be used as soon as they are available. The task heterogeneity and timeliness requirements make simulations hard to support efficiently in BSP-style systems. To demonstrate, we compare (1) an MPI implementation that submits $3n$ parallel simulation runs on n cores in 3 rounds, with a global barrier between rounds[§], to (2) a Ray program that issues the same $3n$ tasks while concurrently gathering simulation results back to the driver. Table 4 shows that both systems scale well, yet Ray achieves up to 1.8$\times$ throughput. This motivates a programming model that can dynamically spawn and collect the results of fine-grained simulation tasks.

System, programming model	1 CPU	16 CPUs	256 CPUs
MPI, bulk synchronous	22.6K	208K	2.16M
Ray, asynchronous tasks	22.3K	290K	4.03M

Table 4: Timesteps per second for the Pendulum-v0 simulator in OpenAI Gym [13]. Ray allows for better utilization when running heterogeneous simulations at scale.

[§]Note that experts *can* use MPI's asynchronous primitives to get around barriers—at the expense of increased program complexity —we nonetheless chose such an implementation to simulate BSP.

5.3 RL Applications

Without a system that can tightly couple the training, simulation, and serving steps, reinforcement learning algorithms today are implemented as one-off solutions that make it difficult to incorporate optimizations that, for example, require a different computation structure or that utilize different architectures. Consequently, with implementations of two representative reinforcement learning applications in Ray, we are able to match and even outperform custom systems built specifically for these algorithms. The primary reason is the flexibility of Ray's programming model, which can express application-level optimizations that would require substantial engineering effort to port to custom-built systems, but are transparently supported by Ray's dynamic task graph execution engine.

5.3.1 Evolution Strategies

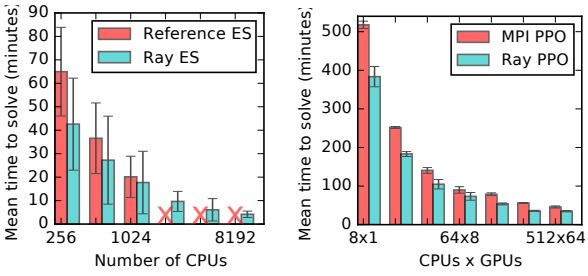
To evaluate Ray on large-scale RL workloads, we implement the evolution strategies (ES) algorithm and compare to the reference implementation [49]—a system specially built for this algorithm that relies on Redis for messaging and low-level multiprocessing libraries for data-sharing. The algorithm periodically broadcasts a new policy to a pool of workers and aggregates the results of roughly 10000 tasks (each performing 10 to 1000 simulation steps).

As shown in Figure 14a, an implementation on Ray scales to 8192 cores. Doubling the cores available yields an average completion time speedup of 1.6×. Conversely, the special-purpose system fails to complete at 2048 cores, where the work in the system exceeds the processing capacity of the application driver. To avoid this issue, the Ray implementation uses an aggregation tree of actors, reaching a median time of 3.7 minutes, more than twice as fast as the best published result (10 minutes).

Initial parallelization of a serial implementation using Ray required modifying only 7 lines of code. Performance improvement through hierarchical aggregation was easy to realize with Ray's support for nested tasks and actors. In contrast, the reference implementation had several hundred lines of code dedicated to a protocol for communicating tasks and data between workers, and would require further engineering to support optimizations like hierarchical aggregation.

5.3.2 Proximal Policy Optimization

We implement Proximal Policy Optimization (PPO) [51] in Ray and compare to a highly-optimized reference implementation [5] that uses OpenMPI communication primitives. The algorithm is an asynchronous scatter-gather, where new tasks are assigned to simulation actors as they return rollouts to the driver. Tasks are submitted until 320000 simulation steps are collected (each task produces between 10 and 1000 steps). The policy update performs 20 steps of SGD with a batch size of 32768. The model parameters in this example are roughly 350KB. These experiments were run using p2.16xlarge (GPU) and m4.16xlarge (high CPU) instances.



(a) Evolution Strategies (b) PPO

Figure 14: Time to reach a score of 6000 in the Humanoid-v1 task [13]. **(a)** The Ray ES implementation scales well to 8192 cores and achieves a median time of 3.7 minutes, over twice as fast as the best published result. The special-purpose system failed to run beyond 1024 cores. ES is faster than PPO on this benchmark, but shows greater runtime variance. **(b)** The Ray PPO implementation outperforms a specialized MPI implementation [5] with fewer GPUs, at a fraction of the cost. The MPI implementation required 1 GPU for every 8 CPUs, whereas the Ray version required at most 8 GPUs (and never more than 1 GPU per 8 CPUs).

As shown in Figure 14b, the Ray implementation outperforms the optimized MPI implementation in all experiments, while using a fraction of the GPUs. The reason is that Ray is heterogeneity-aware and allows the user to utilize asymmetric architectures by expressing resource requirements at the granularity of a task or actor. The Ray implementation can then leverage TensorFlow's single-process multi-GPU support and can pin objects in GPU memory when possible. This optimization cannot be easily ported to MPI due to the need to asynchronously gather rollouts to a single GPU process. Indeed, [5] includes two custom implementations of PPO, one using MPI for large clusters and one that is optimized for GPUs but that is restricted to a single node. Ray allows for an implementation suitable for both scenarios.

Ray's ability to handle resource heterogeneity also decreased PPO's cost by a factor of 4.5 [4], since CPU-only tasks can be scheduled on cheaper high-CPU instances. In contrast, MPI applications often exhibit symmetric architectures, in which all processes run the same code and require identical resources, in this case preventing the use of CPU-only machines for scale-out. Furthermore, the MPI implementation requires on-demand instances since it does not transparently handle failure. Assuming 4× cheaper spot instances, *Ray's fault tolerance and resource-aware scheduling together cut costs by 18×*.

6 Related Work

Dynamic task graphs. Ray is closely related to CIEL [40] and Dask [48]. All three support dynamic task graphs with nested tasks and implement the futures abstraction. CIEL also provides lineage-based fault tolerance, while Dask, like Ray, fully integrates with Python. However, Ray differs in two aspects that have important performance consequences. First, Ray extends the task model with an actor abstraction. This is necessary for efficient stateful computation in distributed training and serving, to keep the model data collocated with the computation. Second, Ray employs a fully distributed and decoupled control plane and scheduler, instead of relying on a single master storing all metadata. This is critical for efficiently supporting primitives like allreduce without system modification. At peak performance for 100MB on 16 nodes, allreduce on Ray (Section 5.1) submits 32 rounds of 16 tasks in 200ms. Meanwhile, Dask reports a maximum scheduler throughput of 3k tasks/s on 512 cores [3]. With a centralized scheduler, each round of allreduce would then incur a minimum of $\sim$5ms of scheduling delay, translating to up to 2$\times$ worse completion time (Figure 12b). Even with a decentralized scheduler, coupling the control plane information with the scheduler leaves the latter on the critical path for data transfer, adding an extra roundtrip to every round of allreduce.

Dataflow systems. Popular dataflow systems, such as MapReduce [20], Spark [65], and Dryad [28] have widespread adoption for analytics and ML workloads, but their computation model is too restrictive for a fine-grained and dynamic simulation workload. Spark and MapReduce implement the BSP execution model, which assumes that tasks within the same stage perform the same computation and take roughly the same amount of time. Dryad relaxes this restriction but lacks support for dynamic task graphs. Furthermore, none of these systems provide an actor abstraction, nor implement a distributed scalable control plane and scheduler. Finally, Naiad [39] is a dataflow system that provides improved scalability for some workloads, but only supports static task graphs.

Machine learning frameworks. TensorFlow [7] and MXNet [18] target deep learning workloads and efficiently leverage both CPUs and GPUs. While they achieve great performance for training workloads consisting of static DAGs of linear algebra operations, they have limited support for the more general computation required to tightly couple training with simulation and embedded serving. TensorFlow Fold [33] provides some support for dynamic task graphs, as well as MXNet through its internal C++ APIs, but neither fully supports the ability to modify the DAG during execution in response to task progress, task completion times, or faults. TensorFlow and MXNet in principle achieve generality by allowing the programmer to simulate low-level message-passing and synchronization primitives, but the pitfalls and user experience in this case are similar to those of MPI. OpenMPI [22] can achieve high performance, but it is relatively hard to program as it requires explicit coordination to handle heterogeneous and dynamic task graphs. Furthermore, it forces the programmer to explicitly handle fault tolerance.

Actor systems. Orleans [14] and Akka [1] are two actor frameworks well suited to developing highly available and concurrent distributed systems. However, compared to Ray, they provide less support for recovery from data loss. To recover *stateful actors*, the Orleans developer must explicitly checkpoint actor state and intermediate responses. *Stateless actors* in Orleans can be replicated for scale-out, and could therefore act as tasks, but unlike in Ray, they have no lineage. Similarly, while Akka explicitly supports persisting actor state across failures, it does not provide efficient fault tolerance for *stateless computation* (i.e., tasks). For message delivery, Orleans provides at-least-once and Akka provides at-most-once semantics. In contrast, Ray provides transparent fault tolerance and exactly-once semantics, as each method call is logged in the GCS and both arguments and results are immutable. We find that in practice these limitations do not affect the performance of our applications. Erlang [10] and C++ Actor Framework [17], two other actor-based systems, have similarly limited support for fault tolerance.

Global control store and scheduling. The concept of logically centralizing the control plane has been previously proposed in software defined networks (SDNs) [16], distributed file systems (e.g., GFS [23]), resource management (e.g., Omega [52]), and distributed frameworks (e.g., MapReduce [20], BOOM [9]), to name a few. Ray draws inspiration from these pioneering efforts, but provides significant improvements. In contrast with SDNs, BOOM, and GFS, Ray decouples the storage of the control plane information (e.g., GCS) from the logic implementation (e.g., schedulers). This allows both storage and computation layers to scale independently, which is key to achieving our scalability targets. Omega uses a distributed architecture in which schedulers coordinate via globally shared state. To this architecture, Ray adds global schedulers to balance load across local schedulers, and targets ms-level, not second-level, task scheduling.

Ray implements a unique distributed bottom-up scheduler that is horizontally scalable, and can handle dynamically constructed task graphs. Unlike Ray, most existing cluster computing systems [20, 64, 40] use a centralized scheduler architecture. While Sparrow [45] is decentralized, its schedulers make independent decisions, limiting the possible scheduling policies, and all tasks of a job are handled by the same global scheduler. Mesos [26] implements a two-level hierarchical scheduler, but its top-level scheduler manages frameworks, not individual tasks.

Canary [47] achieves impressive performance by having each scheduler instance handle a portion of the task graph, but does not handle dynamic computation graphs.

Cilk [12] is a parallel programming language whose work-stealing scheduler achieves provably efficient load-balancing for dynamic task graphs. However, with no central coordinator like Ray's global scheduler, this fully parallel design is also difficult to extend to support data locality and resource heterogeneity in a distributed setting.

7 Discussion and Experiences

Building Ray has been a long journey. It started two years ago with a Spark library to perform distributed training and simulations. However, the relative inflexibility of the BSP model, the high per-task overhead, and the lack of an actor abstraction led us to develop a new system. Since we released Ray roughly one year ago, several hundreds of people have used it and several companies are running it in production. Here we discuss our experience developing and using Ray, and some early user feedback.

API. In designing the API, we have emphasized minimalism. Initially we started with a basic *task* abstraction. Later, we added the **wait()** primitive to accommodate rollouts with heterogeneous durations and the *actor* abstraction to accommodate third-party simulators and amortize the overhead of expensive initializations. While the resulting API is relatively low-level, it has proven both powerful and simple to use. We have already used this API to implement many state-of-the-art RL algorithms on top of Ray, including A3C [36], PPO [51], DQN [37], ES [49], DDPG [55], and Ape-X [27]. In most cases it took us just a few tens of lines of code to port these algorithms to Ray. Based on early user feedback, we are considering enhancing the API to include higher level primitives and libraries, which could also inform scheduling decisions.

Limitations. Given the workload generality, specialized optimizations are hard. For example, we must make scheduling decisions without full knowledge of the computation graph. Scheduling optimizations in Ray might require more complex runtime profiling. In addition, storing lineage for each task requires the implementation of garbage collection policies to bound storage costs in the GCS, a feature we are actively developing.

Fault tolerance. We are often asked if fault tolerance is really needed for AI applications. After all, due to the statistical nature of many AI algorithms, one could simply ignore failed rollouts. Based on our experience, our answer is "yes". First, the ability to ignore failures makes applications much easier to write and reason about. Second, our particular implementation of fault tolerance via deterministic replay dramatically simplifies debugging as it allows us to easily reproduce most errors. This is particularly important since, due to their stochasticity, AI algorithms are notoriously hard to debug. Third, fault tolerance helps save money since it allows us to run on cheap resources like spot instances on AWS. Of course, this comes at the price of some overhead. However, we found this overhead to be minimal for our target workloads.

GCS and Horizontal Scalability. The GCS dramatically simplified Ray development and debugging. It enabled us to query the entire system state while debugging Ray itself, instead of having to manually expose internal component state. In addition, the GCS is also the backend for our timeline visualization tool, used for application-level debugging.

The GCS was also instrumental to Ray's horizontal scalability. In Section 5, we were able to scale by adding more shards whenever the GCS became a bottleneck. The GCS also enabled the global scheduler to scale by simply adding more replicas. Due to these advantages, we believe that centralizing control state will be a key design component of future distributed systems.

8 Conclusion

No general-purpose system today can efficiently support the tight loop of training, serving, and simulation. To express these core building blocks and meet the demands of emerging AI applications, Ray unifies task-parallel and actor programming models in a single dynamic task graph and employs a scalable architecture enabled by the global control store and a bottom-up distributed scheduler. The programming flexibility, high throughput, and low latencies simultaneously achieved by this architecture is particularly important for emerging artificial intelligence workloads, which produce tasks diverse in their resource requirements, duration, and functionality. Our evaluation demonstrates linear scalability up to 1.8 million tasks per second, transparent fault tolerance, and substantial performance improvements on several contemporary RL workloads. Thus, Ray provides a powerful combination of flexibility, performance, and ease of use for the development of future AI applications.

9 Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1730628 and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware as well as by NSF grant DGE-1106400. We are grateful to our anonymous reviewers and our shepherd, Miguel Castro, for thoughtful feedback, which helped improve the quality of this paper.

References

[1] Akka. https://akka.io/.

[2] Apache Arrow. https://arrow.apache.org/.

[3] Dask Benchmarks. http://matthewrocklin.com/blog/work/2017/07/03/scaling.

[4] EC2 Instance Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[5] OpenAI Baselines: high-quality implementations of reinforcement learning algorithms. https://github.com/openai/baselines.

[6] TensorFlow Serving. https://www.tensorflow.org/serving/.

[7] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA* (2016).

[8] AGARWAL, A., BIRD, S., COZOWICZ, M., HOANG, L., LANGFORD, J., LEE, S., LI, J., MELAMED, D., OSHRI, G., RIBAS, O., SEN, S., AND SLIVKINS, A. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966* (2016).

[9] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM Analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 223–236.

[10] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. Concurrent programming in ERLANG.

[11] BEATTIE, C., LEIBO, J. Z., TEPLYASHIN, D., WARD, T., WAINWRIGHT, M., KÜTTLER, H., LEFRANCQ, A., GREEN, S., VALDÉS, V., SADIK, A., ET AL. DeepMind Lab. *arXiv preprint arXiv:1612.03801* (2016).

[12] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[13] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., AND ZAREMBA, W. OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).

[14] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 16.

[15] CARBONE, P., EWEN, S., FÓRA, G., HARIDI, S., RICHTER, S., AND TZOUMAS, K. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1718–1729.

[16] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev. 37*, 4 (Aug. 2007), 1–12.

[17] CHAROUSSET, D., SCHMIDT, T. C., HIESGEN, R., AND WÄHLISCH, M. Native actors: A scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control* (2013), ACM, pp. 87–96.

[18] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems (LearningSys'16)* (2016).

[19] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 613–627.

[20] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[21] DENNIS, J. B., AND MISUNAS, D. P. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture* (New York, NY, USA, 1975), ISCA '75, ACM, pp. 126–132.

[22] GABRIEL, E., FAGG, G. E., BOSILCA, G., ANGSKUN, T., DONGARRA, J. J., SQUYRES, J. M., SAHAY, V., KAMBADUR, P., BARRETT, B., LUMSDAINE, A., CASTAIN, R. H., DANIEL, D. J., GRAHAM, R. L., AND WOODALL, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[23] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. 29–43.

[24] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 599–613.

[25] GU*, S., HOLLY*, E., LILLICRAP, T., AND LEVINE, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA 2017)* (2017).

[26] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[27] HORGAN, D., QUAN, J., BUDDEN, D., BARTH-MARON, G., HESSEL, M., VAN HASSELT, H., AND SILVER, D. Distributed prioritized experience replay. *International Conference on Learning Representations* (2018).

[28] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[29] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[30] JORDAN, M. I., AND MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. *Science 349*, 6245 (2015), 255–260.

[31] LEIBIUSKY, J., EISBRUCH, G., AND SIMONASSI, D. *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[32] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, pp. 583–598.

[33] LOOKS, M., HERRESHOFF, M., HUTCHINS, D., AND NORVIG, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[34] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. GraphLab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (Arlington, Virginia, United States, 2010), UAI'10, pp. 340–349.

[35] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[36] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T. P., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning* (2016).

[37] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[38] MURRAY, D. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.

[39] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[40] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[41] NAIR, A., SRINIVASAN, P., BLACKWELL, S., ALCICEK, C., FEARON, R., MARIA, A. D., PANNEERSHELVAM, V., SULEYMAN, M., BEATTIE, C., PETERSEN, S., LEGG, S., MNIH, V., KAVUKCUOGLU, K., AND SILVER, D. Massively parallel methods for deep reinforcement learning, 2015.

[42] NG, A., COATES, A., DIEL, M., GANAPATHI, V., SCHULTE, J., TSE, B., BERGER, E., AND LIANG, E. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX* (2006), 363–372.

[43] NISHIHARA, R., MORITZ, P., WANG, S., TUMANOV, A., PAUL, W., SCHLEIER-SMITH, J., LIAW, R., NIKNAMI, M., JORDAN, M. I., AND STOICA, I. Real-time machine learning: The missing pieces. In *Workshop on Hot Topics in Operating Systems* (2017).

[44] OPENAI. OpenAI Dota 2 1v1 bot. `https://openai.com/the-international/`, 2017.

[45] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[46] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in PyTorch.

[47] QU, H., MASHAYEKHI, O., TEREI, D., AND LEVIS, P. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412* (2016).

[48] ROCKLIN, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), K. Huff and J. Bergstra, Eds., pp. 130 – 136.

[49] SALIMANS, T., HO, J., CHEN, X., AND SUTSKEVER, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

[50] SANFILIPPO, S. Redis: An open source, in-memory data structure store. `https://redis.io/`, 2009.

[51] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[52] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 351–364.

[53] SERGEEV, A., AND DEL BALSO, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799* (2018).

[54] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[55] SILVER, D., LEVER, G., HEESS, N., DEGRIS, T., WIERSTRA, D., AND RIEDMILLER, M. Deterministic policy gradient algorithms. In *ICML* (2014).

[56] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*. MIT press Cambridge, 1998.

[57] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[58] TIAN, Y., GONG, Q., SHANG, W., WU, Y., AND ZITNICK, C. L. ELF: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)* (2017).

[59] TODOROV, E., EREZ, T., AND TASSA, Y. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on* (2012), IEEE, pp. 5026–5033.

[60] VAN DEN BERG, J., MILLER, S., DUCKWORTH, D., HU, H., WAN, A., FU, X.-Y., GOLDBERG, K., AND ABBEEL, P. Superhuman performance of surgical tasks by robots using iterative learning from human-guided demonstrations. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on* (2010), IEEE, pp. 2074–2081.

[61] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association.

[62] VENKATARAMAN, S., PANDA, A., OUSTERHOUT, K., GHODSI, A., ARMBRUST, M., RECHT, B., FRANKLIN, M., AND STOICA, I. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles* (2017), SOSP '17, ACM.

[63] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[64] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[65] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache Spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen[1], Thierry Moreau[1], Ziheng Jiang[1,2], Lianmin Zheng[3], Eddie Yan[1]

Meghan Cowan[1], Haichen Shen[1], Leyuan Wang[4,2], Yuwei Hu[5], Luis Ceze[1], Carlos Guestrin[1], Arvind Krishnamurthy[1]
[1]Paul G. Allen School of Computer Science & Engineering, University of Washington

[2] AWS, [3]Shanghai Jiao Tong University, [4]UC Davis, [5]Cornell

Abstract

There is an increasing need to bring machine learning to a wide diversity of hardware devices. Current frameworks rely on vendor-specific operator libraries and optimize for a narrow range of server-class GPUs. Deploying workloads to new platforms – such as mobile phones, embedded devices, and accelerators (e.g., FPGAs, ASICs) – requires significant manual effort. We propose TVM, a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also automates optimization of low-level programs to hardware characteristics by employing a novel, learning-based cost modeling method for rapid exploration of code optimizations. Experimental results show that TVM delivers performance across hardware back-ends that are competitive with state-of-the-art, hand-tuned libraries for low-power CPU, mobile GPU, and server-class GPUs. We also demonstrate TVM's ability to target new accelerator back-ends, such as the FPGA-based generic deep learning accelerator. The system is open sourced and in production use inside several major companies.

1 Introduction

Deep learning (DL) models can now recognize images, process natural language, and defeat humans in challenging strategy games. There is a growing demand to deploy smart applications to a wide spectrum of devices, ranging from cloud servers to self-driving cars and embedded devices. Mapping DL workloads to these devices is complicated by the diversity of hardware characteristics, including embedded CPUs, GPUs, FPGAs, and ASICs (e.g., the TPU [21]). These hardware targets diverge in terms of memory organization, compute functional units, etc., as shown in Figure 1.

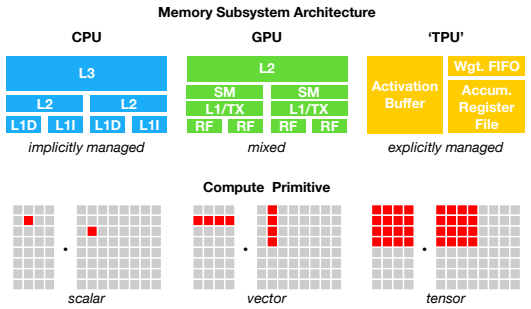


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

Current DL frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch, rely on a computational graph intermediate representation to implement optimizations, e.g., auto differentiation and dynamic memory management [3, 4, 9]. Graph-level optimizations, however, are often too high-level to handle hardware back-end-specific operator-level transformations. Most of these frameworks focus on a narrow class of server-class GPU devices and delegate target-specific optimizations to highly engineered and vendor-specific operator libraries. These operator-level libraries require significant manual tuning and hence are too specialized and opaque to be easily ported across hardware devices. Providing support in various DL frameworks for diverse hardware back-ends presently requires significant engineering effort. Even for supported back-ends, frameworks must make the difficult choice between: (1) avoiding graph optimizations that yield new operators not in the predefined operator library, and (2) using unoptimized implementations of these new operators.

To enable both graph- and operator-level optimiza-

tions for diverse hardware back-ends, we take a fundamentally different, end-to-end approach. *We built TVM, a compiler that takes a high-level specification of a deep learning program from existing frameworks and generates low-level optimized code for a diverse set of hardware back-ends.* To be attractive to users, TVM needs to offer performance competitive with the multitude of manually optimized operator libraries across diverse hardware back-ends. This goal requires addressing the key challenges described below.

Leveraging Specific Hardware Features and Abstractions. DL accelerators introduce optimized tensor compute primitives [1, 12, 21], while GPUs and CPUs continuously improve their processing elements. This poses a significant challenge in generating optimized code for a given operator description. The inputs to hardware instructions are multi-dimensional, with fixed or variable lengths; they dictate different data layouts; and they have special requirements for memory hierarchy. The system must effectively exploit these complex primitives to benefit from acceleration. Further, accelerator designs also commonly favor leaner control [21] and offload most scheduling complexity to the compiler stack. For specialized accelerators, the system now needs to generate code that explicitly controls pipeline dependencies to hide memory access latency – a job that hardware performs for CPUs and GPUs.

Large Search Space for Optimization Another challenge is producing efficient code without manually tuning operators. The combinatorial choices of memory access, threading pattern, and novel hardware primitives creates a huge configuration space for generated code (e.g., loop tiles and ordering, caching, unrolling) that would incur a large search cost if we implement black box auto-tuning. One could adopt a predefined cost model to guide the search, but building an accurate cost model is difficult due to the increasing complexity of modern hardware. Furthermore, such an approach would require us to build separate cost models for each hardware type.

TVM addresses these challenges with three key modules. (**1**) We introduce a *tensor expression language* to build operators and provide program transformation primitives that generate different versions of the program with various optimizations. This layer extends Halide [32]'s compute/schedule separation concept by also separating target hardware intrinsics from transformation primitives, which enables support for novel accelerators and their corresponding new intrinsics. Moreover, we introduce new transformation primitives to address GPU-related challenges and enable deployment to specialized accelerators. We can then apply different sequences of program transformations to form a rich space of valid programs for a given operator declaration. (**2**) We introduce an *automated program optimization framework* to find optimized tensor operators. The optimizer is guided by an ML-based cost model that adapts and improves as we collect more data from a hardware back-end. (**3**) On top of the automatic code generator, we introduce a *graph rewriter* that takes full advantage of high- and operator-level optimizations.

By combining these three modules, TVM can take model descriptions from existing deep learning frameworks, perform joint high- and low-level optimizations, and generate hardware-specific optimized code for back-ends, e.g., CPUs, GPUs, and FPGA-based specialized accelerators.

This paper makes the following contributions:

- We identify the major optimization challenges in providing performance portability to deep learning workloads across diverse hardware back-ends.
- We introduce novel schedule primitives that take advantage of cross-thread memory reuse, novel hardware intrinsics, and latency hiding.
- We propose and implement a machine learning based optimization system to automatically explore and search for optimized tensor operators.
- We build an end-to-end compilation and optimization stack that allows the deployment of deep learning workloads specified in high-level frameworks (including TensorFlow, MXNet, PyTorch, Keras, CNTK) to diverse hardware back-ends (including CPUs, server GPUs, mobile GPUs, and FPGA-based accelerators). The open-sourced TVM is in production use inside several major companies.

  We evaluated TVM using real world workloads on a server-class GPU, an embedded GPU, an embedded CPU, and a custom generic FPGA-based accelerator. Experimental results show that TVM offers portable performance across back-ends and achieves speedups ranging from 1.2× to 3.8× over existing frameworks backed by hand-optimized libraries.

2 Overview

This section describes TVM by using an example to walk through its components. Figure 2 summarizes execution steps in TVM and their corresponding sections in the paper. The system first takes as input a model from an existing framework and transforms it into a computational graph representation. It then performs high-level dataflow rewriting to generate an optimized graph. The operator-level optimization module must generate efficient code for each fused operator in this graph. Operators are specified in a declarative tensor expression lan-

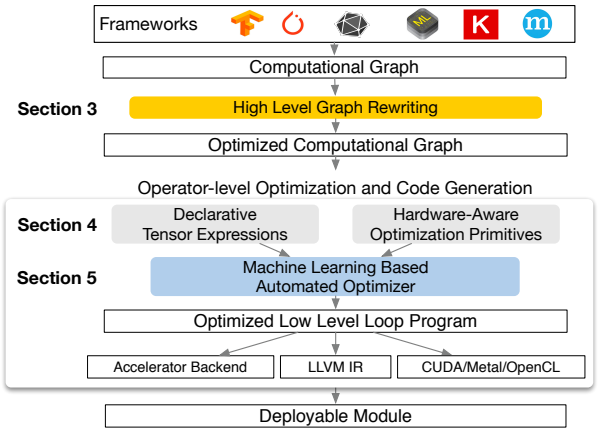


Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

guage; execution details are unspecified. TVM identifies a collection of possible code optimizations for a given hardware target's operators. Possible optimizations form a large space, so we use an ML-based cost model to find optimized operators. Finally, the system packs the generated code into a deployable module.

End-User Example. In a few lines of code, a user can take a model from existing deep learning frameworks and call the TVM API to get a deployable module:

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

This compiled runtime module contains three components: the final optimized computational graph (`graph`), generated operators (`lib`), and module parameters (`params`). These components can then be used to deploy the model to the target back-end:

```
import tvm.runtime as t
module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
```

TVM supports multiple deployment back-ends in languages such as C++, Java and Python. The rest of this paper describes TVM's architecture and how a system programmer can extend it to support new back-ends.

3 Optimizing Computational Graphs

Computational graphs are a common way to represent programs in DL frameworks [3, 4, 7, 9]. Figure 3 shows

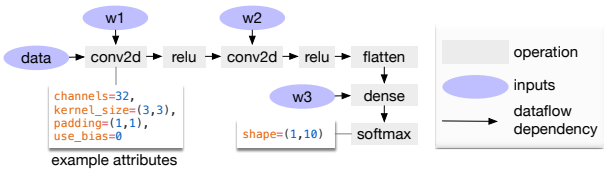


Figure 3: Example computational graph of a two-layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or more tensors. Tensor operations can be parameterized by attributes to configure their behavior (e.g., padding or strides).

an example computational graph representation of a two-layer convolutional neural network. The main difference between this high-level representation and a low-level compiler intermediate representation (IR), such as LLVM, is that the intermediate data items are large, multi-dimensional tensors. Computational graphs provide a global view of operators, but they avoid specifying how each operator must be implemented. Like LLVM IRs, a computational graph can be transformed into functionally equivalent graphs to apply optimizations. We also take advantage of shape specificity in common DL workloads to optimize for a fixed set of input shapes.

TVM exploits a computational graph representation to apply high-level optimizations: a node represents an operation on tensors or program inputs, and edges represent data dependencies between operations. It implements many graph-level optimizations, including: *operator fusion*, which fuses multiple small operations together; *constant-folding*, which pre-computes graph parts that can be determined statically, saving execution costs; a *static memory planning pass*, which pre-allocates memory to hold each intermediate tensor; and *data layout transformations*, which transform internal data layouts into back-end-friendly forms. We now discuss operator fusion and the data layout transformation.

Operator Fusion. Operator fusion combines multiple operators into a single kernel without saving the intermediate results in memory. This optimization can greatly reduce execution time, particularly in GPUs and specialized accelerators. Specifically, we recognize four categories of graph operators: (1) injective (one-to-one map, e.g., add), (2) reduction (e.g., sum), (3) complex-out-fusable (can fuse element-wise map to output, e.g., conv2d), and (4) opaque (cannot be fused, e.g., sort). We provide generic rules to fuse these operators, as follows. Multiple injective operators can be fused into another injective operator. A reduction operator can be fused with input injective operators (e.g., fuse scale and sum). Operators such as conv2d are complex-out-fusable, and we

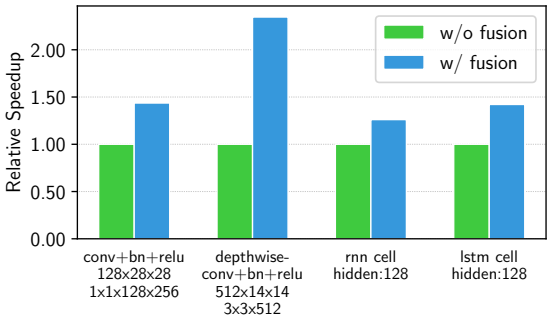


Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

can fuse element-wise operators to its output. We can apply these rules to transform the computational graph into a fused version. Figure 4 demonstrates the impact of this optimization on different workloads. We find that fused operators generate up to a 1.2× to 2× speedup by reducing memory accesses.

Data Layout Transformation. There are multiple ways to store a given tensor in the computational graph. The most common data layout choices are column major and row major. In practice, we may prefer to use even more complicated data layouts. For instance, a DL accelerator might exploit 4×4 matrix operations, requiring data to be tiled into 4×4 chunks to optimize for access locality.

Data layout optimization converts a computational graph into one that can use better internal data layouts for execution on the target hardware. It starts by specifying the preferred data layout for each operator given the constraints dictated by memory hierarchies. We then perform the proper layout transformation between a producer and a consumer if their preferred data layouts do not match.

While high-level graph optimizations can greatly improve the efficiency of DL workloads, they are only as effective as what the operator library provides. Currently, the few DL frameworks that support operator fusion require the operator library to provide an implementation of the fused patterns. With more network operators introduced on a regular basis, the number of possible fused kernels can grow dramatically. This approach is no longer sustainable when targeting an increasing number of hardware back-ends since the required number of fused pattern implementations grows combinatorially with the number of data layouts, data types, and accelerator intrinsics that must be supported. It is not feasible to handcraft operator kernels for the various operations desired by a program and for each back-end. To

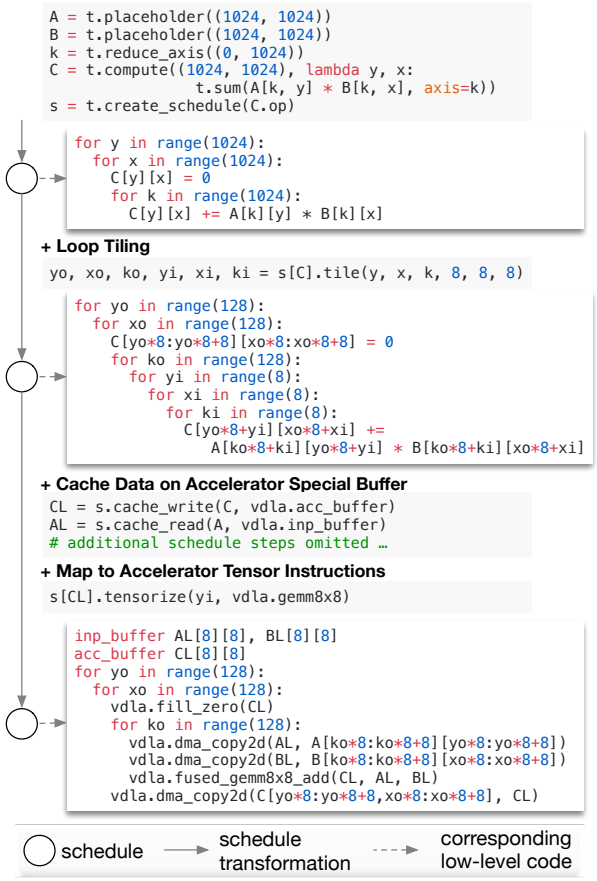


Figure 5: Example schedule transformations that optimize a matrix multiplication on a specialized accelerator.

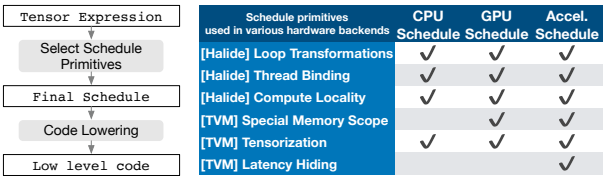
this end, we next propose a code generation approach that can generate various possible implementations for a given model's operators.

4 Generating Tensor Operations

TVM produces efficient code for each operator by generating many valid implementations on each hardware back-end and choosing an optimized implementation. This process builds on Halide's idea of decoupling descriptions from computation rules (or *schedule optimizations*) [32] and extends it to support new optimizations (nested parallelism, tensorization, and latency hiding) and a wide array of hardware back-ends. We now highlight TVM-specific features.

4.1 Tensor Expression and Schedule Space

We introduce a tensor expression language to support automatic code generation. Unlike high-level computation graph representations, where the implementation of tensor operations is opaque, each operation is described in



Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

Figure 6: TVM schedule lowering and code generation process. The table lists existing Halide and novel TVM scheduling primitives being used to optimize schedules for CPUs, GPUs and accelerator back-ends. Tensorization is essential for accelerators, but it can also be used for CPUs and GPUs. Special memory-scope enables memory reuse in GPUs and explicit management of on-chip memory in accelerators. Latency hiding is specific to TPU-like accelerators.

an index formula expression language. The following code shows an example tensor expression to compute transposed matrix multiplication:

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
      t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

Each compute operation specifies both the shape of the output tensor and an expression describing how to compute each element of it. Our tensor expression language supports common arithmetic and math operations and covers common DL operator patterns. The language does not specify the loop structure and many other execution details, and it provides flexibility for adding hardware-aware optimizations for various back-ends. Adopting the decoupled compute/schedule principle from Halide [32], we use a schedule to denote a specific mapping from a tensor expression to low-level code. Many possible schedules can perform this function.

We build a schedule by incrementally applying basic transformations (schedule primitives) that preserve the program's logical equivalence. Figure 5 shows an example of scheduling matrix multiplication on a specialized accelerator. Internally, TVM uses a data structure to keep track of the loop structure and other information as we apply schedule transformations. This information can then help generate low-level code for a given final schedule.

Our tensor expression takes cues from Halide [32], Darkroom [17], and TACO [23]. Its primary enhancements include support for the new schedule optimizations discussed below. To achieve high performance on many back-ends, we must support enough schedule primitives to cover a diverse set of optimizations on different hardware back-ends. Figure 6 summarizes the operation code generation process and schedule primitives that TVM supports. We reuse helpful primitives and the low-level loop program AST from Halide, and we introduce new primitives to optimize GPU and accelerator performance. The new primitives are necessary to achieve optimal GPU performance and essential for accelerators. CPU, GPU, TPU-like accelerators are three important types of hardware for deep learning. This section describes new optimization primitives for CPUs, GPUs and TPU-like accelerators, while section 5 explains how to automatically derive efficient schedules.

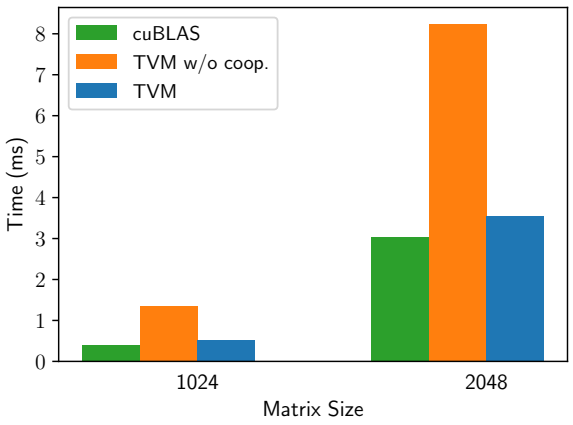


Figure 7: Performance comparison between TVM with and without cooperative shared memory fetching on matrix multiplication workloads. Tested on an NVIDIA Titan X.

4.2 Nested Parallelism with Cooperation

Parallelism is key to improving the efficiency of compute-intensive kernels in DL workloads. Modern GPUs offer massive parallelism, requiring us to bake parallel patterns into schedule transformations. Most existing solutions adopt a model called *nested parallelism*, a form of fork–join. This model requires a parallel schedule primitive to parallelize a data parallel task; each task can be further recursively subdivided into subtasks to exploit the target architecture's multi-level thread hierarchy (e.g., thread groups in GPU). We call this model *shared-nothing nested parallelism* because one working thread cannot look at the data of its sibling within the same parallel computation stage.

An alternative to the shared-nothing approach is to fetch data cooperatively. Specifically, groups of threads can cooperatively fetch the data they all need and place it into a shared memory space. [1] This optimization can take advantage of the GPU memory hierarchy and en-

[1] Halide recently added shared memory support but without general memory scope for accelerators.

able data reuse across threads through shared memory regions. TVM supports this well-known GPU optimization using a schedule primitive to achieve optimal performance. The following GPU code example optimizes matrix multiplication.

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
    for yi in range(8):
      for xi in range(8):
        C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

Figure 7 demonstrates the impact of this optimization. We introduce the concept of *memory scopes* to the schedule space so that a compute stage (AS and BS in the code) can be marked as shared. Without explicit memory scopes, automatic scope inference will mark compute stages as thread-local. The shared task must compute the dependencies of all working threads in the group. Additionally, memory synchronization barriers must be properly inserted to guarantee that shared loaded data is visible to consumers. Finally, in addition to being useful to GPUs, memory scopes let us tag special memory buffers and create special lowering rules when targeting specialized DL accelerators.

4.3 Tensorization

DL workloads have high arithmetic intensity, which can typically be decomposed into tensor operators like matrix-matrix multiplication or 1D convolution. These natural decompositions have led to the recent trend of adding tensor compute primitives [1, 12, 21]. These new primitives create both opportunities and challenges for schedule-based compilation; while using them can improve performance, the compilation framework must seamlessly integrate them. We dub this *tensorization*: it is analogous to vectorization for SIMD architectures but has significant differences. Instruction inputs are multi-dimensional, with fixed or variable lengths, and each has different data layouts. More importantly, we cannot support a fixed set of primitives since new accelerators are emerging with their own variations of tensor instructions. We therefore need an *extensible* solution.

We make tensorization extensible by separating the target hardware intrinsic from the schedule with a mechanism for tensor-intrinsic declaration. We use the same tensor expression language to declare both the behavior of each new hardware intrinsic and the lowering rule associated with it. The following code shows how to declare an 8×8 tensor hardware intrinsic.

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr("r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

Additionally, we introduce a *tensorize* schedule primitive to replace a unit of computation with the corresponding intrinsics. The compiler matches the computation pattern with a hardware declaration and lowers it to the corresponding hardware intrinsic.

Tensorization decouples the schedule from specific hardware primitives, making it easy to extend TVM to support new hardware architectures. The generated code of tensorized schedules aligns with practices in high-performance computing: break complex operations into a sequence of micro-kernel calls. We can also use the *tensorize* primitive to take advantage of handcrafted micro-kernels, which can be beneficial in some platforms. For example, we implement ultra low precision operators for mobile CPUs that operate on data types that are one- or two-bits wide by leveraging a bit-serial matrix vector multiplication micro-kernel. This micro-kernel accumulates results into progressively larger data types to minimize the memory footprint. Presenting the micro-kernel as a tensor intrinsic to TVM yields up to a $1.5\times$ speedup over the non-tensorized version.

4.4 Explicit Memory Latency Hiding

Latency hiding refers to the process of overlapping memory operations with computation to maximize utilization of memory and compute resources. It requires different strategies depending on the target hardware back-end. On CPUs, memory latency hiding is achieved implicitly with simultaneous multithreading [14] or hardware prefetching [10, 20]. GPUs rely on rapid context switching of many warps of threads [44]. In contrast, specialized DL accelerators such as the TPU [21] usually favor leaner control with a *decoupled access-execute* (DAE) architecture [35] and offload the problem of fine-grained synchronization to software.

Figure 9 shows a DAE hardware pipeline that reduces runtime latency. Compared to a monolithic hardware design, the pipeline can hide most memory access overheads and almost fully utilize compute resources. To achieve higher utilization, the instruction stream must be augmented with fine-grained synchronization operations. Without them, dependencies cannot be enforced, leading to erroneous execution. Consequently, DAE hardware pipelines require fine-grained dependence enqueuing/dequeuing operations between the pipeline stages to guar-

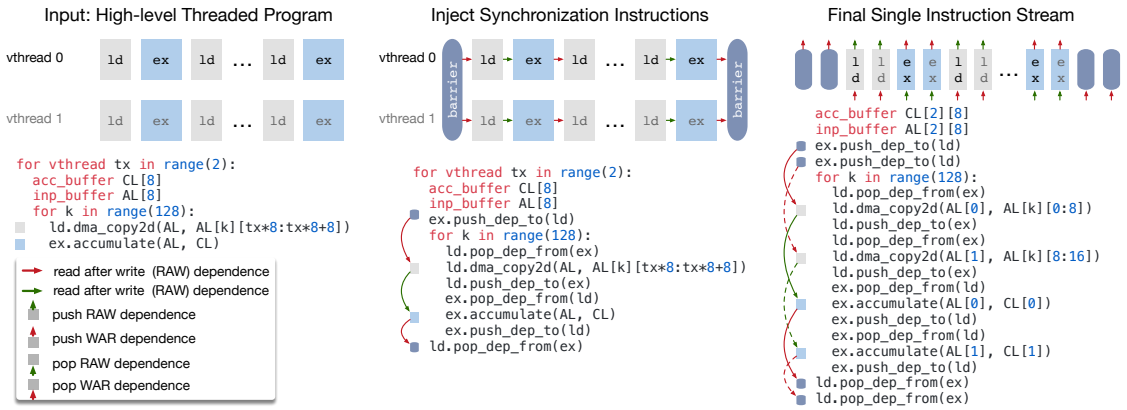


Figure 8: TVM virtual thread lowering transforms a virtual thread-parallel program to a single instruction stream; the stream contains explicit low-level synchronizations that the hardware can interpret to recover the pipeline parallelism required to hide memory access latency.

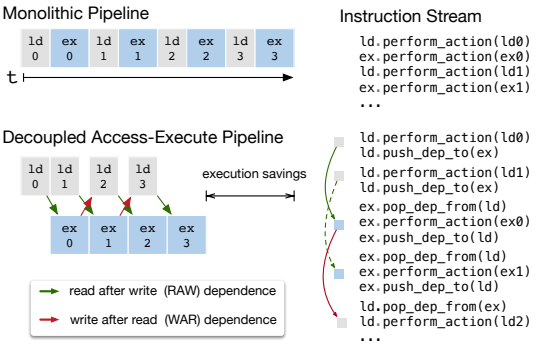


Figure 9: Decoupled Access-Execute in hardware hides most memory access latency by allowing memory and computation to overlap. Execution correctness is enforced by low-level synchronization in the form of dependence token enqueueing/dequeuing actions, which the compiler stack must insert in the instruction stream.

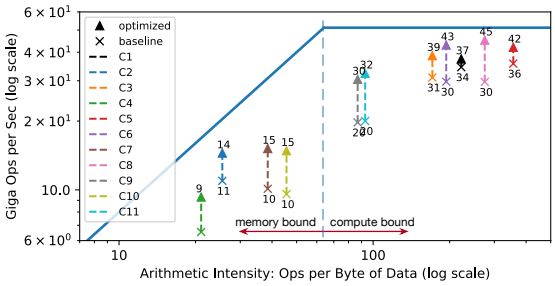


Figure 10: Roofline [47] of an FPGA-based DL accelerator running ResNet inference. With latency hiding enabled by TVM, performance of the benchmarks is brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.

antee correct execution, as shown in Figure 9's instruction stream.

Programming DAE accelerators that require explicit low-level synchronization is difficult. To reduce the programming burden, we introduce a virtual threading scheduling primitive that lets programmers specify a high-level data parallel program as they would a hardware back-end with support for multithreading. TVM then automatically lowers the program to a single instruction stream with low-level explicit synchronization, as shown in Figure 8. The algorithm starts with a high-level multi-threaded program schedule and then inserts the necessary low-level synchronization operations to guarantee correct execution within each thread. Next, it interleaves operations of all virtual threads into a single instruction stream. Finally, the hardware recovers the available pipeline parallelism dictated by the low-level synchronizations in the instruction stream.

Hardware Evaluation of Latency Hiding. We now demonstrate the effectiveness of latency hiding on a custom FPGA-based accelerator design, which we describe in depth in subsection 6.4. We ran each layer of ResNet on the accelerator and used TVM to generate two schedules: one with latency hiding, and one without. The schedule with latency hiding parallelized the program with virtuals threads to expose pipeline parallelism and therefore hide memory access latency. Results are shown in Figure 10 as a roofline diagram [47]; roofline performance diagrams provide insight into how well a given system uses computation and memory resources for different benchmarks. Overall, latency hiding improved performance on all ResNet layers. Peak compute utilization increased from 70% with no latency hiding to 88% with latency hiding.

5 Automating Optimization

Given the rich set of schedule primitives, our remaining problem is to find optimal operator implementations for each layer of a DL model. Here, TVM creates a specialized operator for the specific input shape and layout associated with each layer. Such specialization offers significant performance benefits (in contrast to handcrafted code that would target a smaller diversity of shapes and layouts), but it also raises automation challenges. The system needs to choose the schedule optimizations – such as modifying the loop order or optimizing for the memory hierarchy – as well as schedule-specific parameters, such as the tiling size and the loop unrolling factor. Such combinatorial choices create a large search space of operator implementations for each hardware back-end. To address this challenge, we built an *automated schedule optimizer* with two main components: a schedule explorer that *proposes* promising new configurations, and a machine learning cost model that *predicts* the performance of a given configuration. This section describes these components and TVM's automated optimization flow (Figure 11).

5.1 Schedule Space Specification

We built a *schedule template specification API* to let a developer declare knobs in the schedule space. The template specification allows incorporation of a developer's domain-specific knowledge, as necessary, when specifying possible schedules. We also created a *generic master template for each hardware back-end* that automatically extracts possible knobs based on the computation description expressed using the tensor expression language. At a high level, we would like to consider as many configurations as possible and let the optimizer manage the selection burden. Consequently, the optimizer must search over *billions* of possible configurations for the real world DL workloads used in our experiments.

5.2 ML-Based Cost Model

One way to find the best schedule from a large configuration space is through blackbox optimization, i.e., auto-tuning. This method is used to tune high performance computing libraries [15, 46]. However, auto-tuning requires many experiments to identify a good configuration.

An alternate approach is to build a predefined cost model to guide the search for a particular hardware back-end instead of running all possibilities and measuring their performance. Ideally, a perfect cost model considers all factors affecting performance: memory access patterns, data reuse, pipeline dependencies, and threading patterns, among others. This approach, unfortunately, is burdensome due to the increasing complexity of modern hardware. Furthermore, every new hardware target requires a new (predefined) cost model.

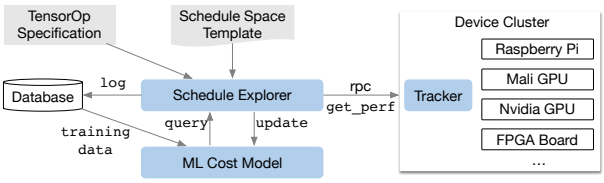


Figure 11: Overview of automated optimization framework. A schedule explorer examines the schedule space using an ML-based cost model and chooses experiments to run on a distributed device cluster via RPC. To improve its predictive power, the ML model is updated periodically using collected data recorded in a database.

Method Category	Data Cost	Model Bias	Need Hardware Info	Learn from History
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

Table 1: Comparison of automation methods. Model bias refers to inaccuracy due to modeling.

We instead take a statistical approach to solve the cost modeling problem. In this approach, a schedule explorer proposes configurations that may improve an operator's performance. For each schedule configuration, we use an ML model that takes the lowered loop program as input and predicts its running time on a given hardware back-end. The model, trained using runtime measurement data collected during exploration, does not require the user to input detailed hardware information. We update the model periodically as we explore more configurations during optimization, which improves accuracy for other related workloads, as well. In this way, the quality of the ML model improves with more experimental trials. Table 1 summarizes the key differences between automation methods. ML-based cost models strike a balance between auto-tuning and predefined cost modeling and can benefit from the historical performance data of related workloads.

Machine Learning Model Design Choices. We must consider two key factors when choosing which ML model the schedule explorer will use: *quality* and *speed*. The schedule explorer queries the cost model frequently, which incurs overheads due to model prediction time and model refitting time. To be useful, these overheads must be smaller than the time it takes to measure per-

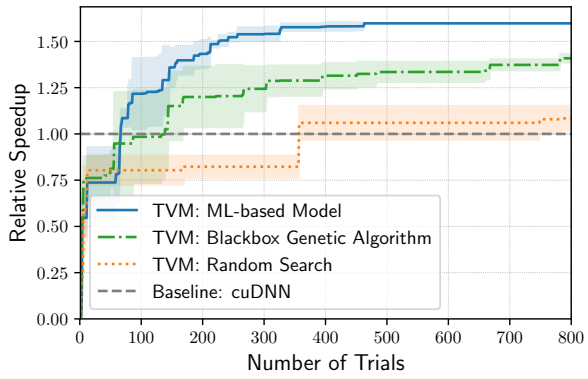


Figure 12: Comparison of different automation methods for a conv2d operator in ResNet-18 on TITAN X. The ML-based model starts with no training data and uses the collected data to improve itself. The Y-axis is the speedup relative to cuDNN. We observe a similar trend for other workloads.

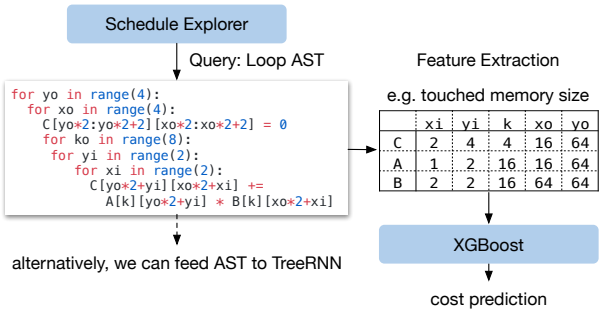


Figure 13: Example workflow of ML cost models. XGBoost predicts costs based on loop program features. TreeRNN directly summarizes the AST.

formance on real hardware, which can be on the order of seconds depending on the specific workload/hardware target. This speed requirement differentiates our problem from traditional hyperparameter tuning problems, where the cost of performing measurements is very high relative to model overheads, and more expensive models can be used. In addition to the choice of model, we need to choose an objective function to train the model, such as the error in a configuration's predicted running time. However, since the explorer selects the top candidates based only on the relative order of the prediction (A runs faster than B), we need not predict the absolute execution times directly. Instead, we use a rank objective to predict the relative order of runtime costs.

We implement several types of models in our ML optimizer. We employ a *gradient tree boosting model* (based on XGBoost [8]), which makes predictions based on features extracted from the loop program; these features include the memory access count and reuse ratio of each memory buffer at each loop level, as well as a one-hot encoding of loop annotations such as "vectorize", "unroll", and "parallel." We also evaluate a *neural network model* that uses TreeRNN [38] to summarize the loop program's AST without feature engineering. Figure 13 summarizes the workflow of the cost models. We found that tree boosting and TreeRNN have similar predictive quality. However, the former performs prediction twice as fast and costs much less time to train. As a result, we chose gradient tree boosting as the default cost model in our experiments. Nevertheless, we believe that both approaches are valuable and expect more future research on this problem.

On average, the tree boosting model does prediction in 0.67 ms, thousands of times faster than running a real measurement. Figure 12 compares an ML-based optimizer to blackbox auto-tuning methods; the former finds better configurations much faster than the latter.

5.3 Schedule Exploration

Once we choose a cost model, we can use it to select promising configurations on which to iteratively run real measurements. In each iteration, the explorer uses the ML model's predictions to select a batch of candidates on which to run the measurements. The collected data is then used as training data to update the model. If no initial training data exists, the explorer picks random candidates to measure.

The simplest exploration algorithm enumerates and runs every configuration through the cost model, selecting the top-k predicted performers. However, this strategy becomes intractable with large search spaces. Instead, we run a parallel simulated annealing algorithm [22]. The explorer starts with random configurations, and, at each step, randomly walks to a nearby configuration. This transition is successful if cost decreases as predicted by the cost model. It is likely to fail (reject) if the target configuration has a higher cost. The random walk tends to converge on configurations that have lower costs as predicted by the cost model. Exploration states persist across cost model updates; we continue from the last configuration after these updates.

5.4 Distributed Device Pool and RPC

A *distributed device pool* scales up the running of on-hardware trials and enables fine-grained resource sharing among multiple optimization jobs. TVM implements a customized, RPC-based distributed device pool that enables clients to run programs on a specific type of device. We can use this interface to compile a program on the host compiler, request a remote device, run the

Name	Operator	H, W	IC, OC	K, S
C1	conv2d	224, 224	3,64	7, 2
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1

Name	Operator	H, W	IC	K, S
D1	depthwise conv2d	112, 112	32	3, 1
D2	depthwise conv2d	112, 112	64	3, 2
D3	depthwise conv2d	56, 56	128	3, 1
D4	depthwise conv2d	56, 56	128	3, 2
D5	depthwise conv2d	28, 28	256	3, 1
D6	depthwise conv2d	28, 28	256	3, 2
D7	depthwise conv2d	14, 14	512	3, 1
D8	depthwise conv2d	14, 14	512	3, 2
D9	depthwise conv2d	7, 7	1024	3, 1

Table 2: Configurations of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet used in the single kernel experiments. H/W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size. All ops use "SAME" padding. All depthwise conv2d operations have channel multipliers of 1.

function remotely, and access results in the same script on the host. TVM's RPC supports dynamic upload and runs cross-compiled modules and functions that use its runtime convention. As a result, the same infrastructure can perform a single workload optimization and end-to-end graph inference. Our approach automates the compile, run, and profile steps across multiple devices. This infrastructure is especially critical for embedded devices, which traditionally require tedious manual effort for cross-compilation, code deployment, and measurement.

6 Evaluation

TVM's core is implemented in C++ (∼50k LoC). We provide language bindings to Python and Java. Earlier sections of this paper evaluated the impact of several individual optimizations and components of TVM, namely, *operator fusion* in Figure 4, *latency hiding* in Figure 10, and the *ML-based cost model* in Figure 12. We now focus on an end-to-end evaluation that aims to answer the following questions:

- Can TVM optimize DL workloads over multiple platforms?
- How does TVM compare to existing DL frameworks (which rely on heavily optimized libraries) on each back-end?
- Can TVM support new, emerging DL workloads (e.g., depthwise convolution, low precision operations)?
- Can TVM support and optimize for new specialized accelerators?

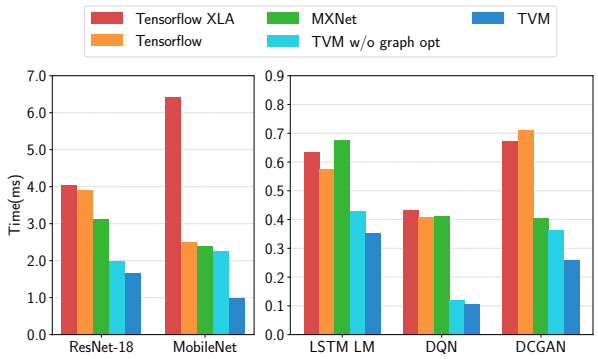


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

To answer these questions, we evaluated TVM on four types of platforms: (1) a server-class GPU, (2) an embedded GPU, (3) an embedded CPU, and (4) a DL accelerator implemented on a low-power FPGA SoC. The benchmarks are based on real world DL inference workloads, including ResNet [16], MobileNet [19], the LSTM Language Model [48], the Deep Q Network (DQN) [28] and Deep Convolutional Generative Adversarial Networks (DCGAN) [31]. We compare our approach to existing DL frameworks, including MxNet [9] and TensorFlow [2], that rely on highly engineered, vendor-specific libraries. TVM performs end-to-end automatic optimization and code generation *without the need for an external operator library*.

6.1 Server-Class GPU Evaluation

We first compared the end-to-end performance of deep neural networks TVM, MXNet (v1.1), Tensorflow (v1.7), and Tensorflow XLA on an Nvidia Titan X. MXNet and Tensorflow both use cuDNN v7 for convolution operators; they implement their own versions of depthwise convolution since it is relatively new and not yet supported by the latest libraries. They also use cuBLAS v8 for matrix multiplications. On the other hand, Tensorflow XLA uses JIT compilation.

Figure 14 shows that TVM outperforms the baselines, with speedups ranging from 1.6× to 3.8× due to both joint graph optimization and the automatic optimizer, which generates high-performance fused opera-

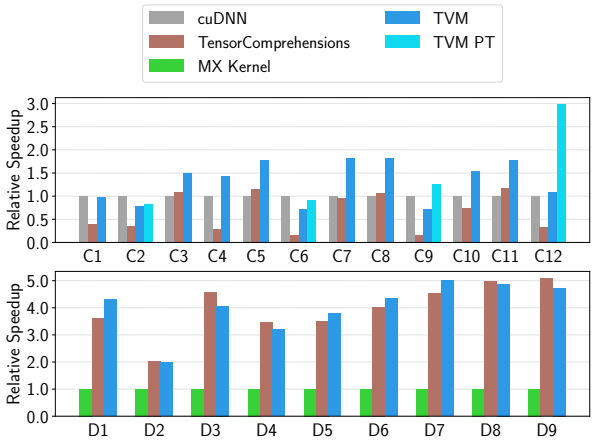


Figure 15: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on a TITAN X. See Table 2 for operator configurations. We also include a weight pre-transformed Winograd [25] for 3x3 conv2d (TVM PT).

tors. DQN's 3.8 x speedup results from its use of unconventional operators (4×4 conv2d, strides=2) that are not well optimized by cuDNN; the ResNet workloads are more conventional. TVM automatically finds optimized operators in both cases.

To evaluate the effectiveness of operator level optimization, we also perform a breakdown comparison for each tensor operator in ResNet and MobileNet, shown in Figure 15. We include TensorComprehension (TC, commit: ef644ba) [42], a recently introduced auto-tuning framework, as an additional baseline. [2] TC results include the best kernels it found in 10 generations × 100 population × 2 random seeds for each operator (i.e., 2000 trials per operator). 2D convolution, one of the most important DL operators, is heavily optimized by cuDNN. However, TVM can still generate better GPU kernels for most layers. Depthwise convolution is a newly introduced operator with a simpler structure [19]. In this case, both TVM and TC can find fast kernels compared to MXNet's handcrafted kernels. TVM's improvements are mainly due to its exploration of a large schedule space and an effective ML-based search algorithm.

6.2 Embedded CPU Evaluation

We evaluated the performance of TVM on an ARM Cortex A53 (Quad Core 1.2GHz). We used Tensorflow Lite (TFLite, commit: 7558b085) as our baseline system. Figure 17 compares TVM operators to hand-optimized

[2] According to personal communication [41], TC is not yet meant to be used for compute-bound problems. However, it is still a good reference baseline to include in the comparison.

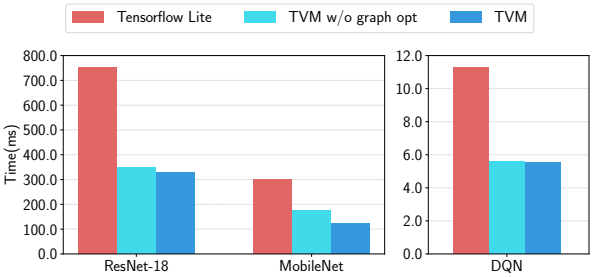


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

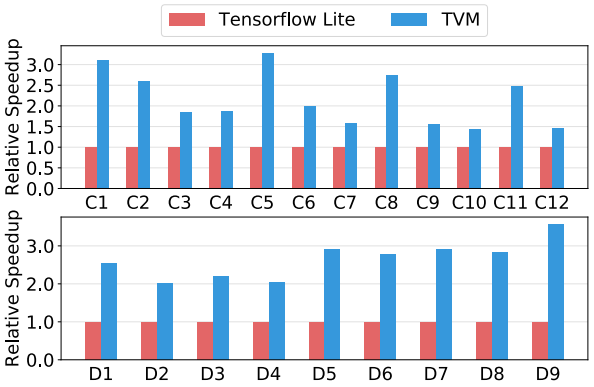


Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

ones for ResNet and MobileNet. We observe that TVM generates operators that outperform the hand-optimized TFLite versions for both neural network workloads. This result also demonstrates TVM's ability to quickly optimize emerging tensor operators, such as depthwise convolution operators. Finally, Figure 16 shows an end-to-end comparison of three workloads, where TVM outperforms the TFLite baseline.[3]

Ultra Low-Precision Operators We demonstrate TVM's ability to support ultra low-precision inference [13, 33] by generating highly optimized operators for fixed-point data types of less than 8-bits. Low-precision networks replace expensive multiplication with vectorized bit-serial multiplication that is composed of bitwise *and* popcount reductions [39]. Achieving efficient low-precision inference requires packing quantized data types into wider standard data types, such as `int8` or `int32`. Our system generates code that outperforms hand-optimized libraries from Caffe2 (commit: 39e07f7)

[3] DCGAN and LSTM results are not presented because they are not yet supported by the baseline.

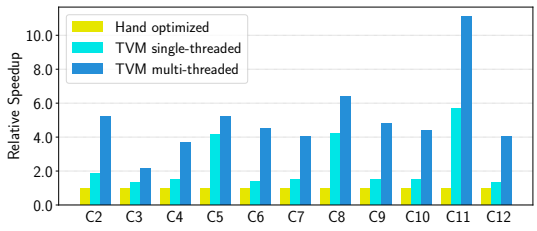


Figure 18: Relative speedup of single- and multi-threaded low-precision conv2d operators in ResNet. Baseline was a single-threaded, hand-optimized implementation from Caffe2 (commit: 39e07f7). C5, C3 are 1x1 convolutions that have less compute intensity, resulting in less speedup by multi-threading.

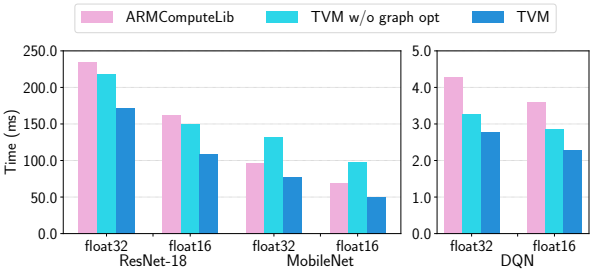


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

[39]. We implemented an ARM-specific *tensorization* intrinsic that leverages ARM instructions to build an efficient, low-precision matrix-vector microkernel.We then used TVM's automated optimizer to explore the scheduling space.

Figure 18 compares TVM to the Caffe2 ultra low-precision library on ResNet for 2-bit activations, 1-bit weights inference. Since the baseline is single threaded, we also compare it to a single-threaded TVM version. Single-threaded TVM outperforms the baseline, particularly for `C5`, `C8`, and `C11` layers; these are convolution layers of kernel size 1×1 and stride of 2 for which the ultra low-precision baseline library is not optimized. Furthermore, we take advantage of additional TVM capabilities to produce a parallel library implementation that shows improvement over the baseline. In addition to the 2-bit+1-bit configuration, TVM can generate and optimize for other precision configurations that are unsupported by the baseline library, offering improved flexibility.

6.3 Embedded GPU Evaluation

For our mobile GPU experiments, we ran our end-to-end pipeline on a Firefly-RK3399 board equipped with an ARM Mali-T860MP4 GPU. The baseline was a vendor-provided library, the ARM Compute Library (v18.03). As shown in Figure 19, we outperformed the baseline on three available models for both `float16` and `float32` (DCGAN and LSTM are not yet supported by the baseline). The speedup ranged from 1.2× to 1.6×.

6.4 FPGA Accelerator Evaluation

Vanilla Deep Learning Accelerator We now relate how TVM tackled accelerator-specific code generation on a generic inference accelerator design we prototyped on an FPGA. We used in this evaluation the Vanilla Deep Learning Accelerator (VDLA) – which distills characteristics from previous accelerator proposals [12, 21, 27] into a minimalist hardware architecture – to demonstrate TVM's ability to generate highly efficient schedules that can target specialized accelerators. Figure 20 shows the high-level hardware organization of the VDLA architecture. VDLA is programmed as a tensor processor to efficiently execute operations with high compute intensity (e.g, matrix multiplication, high dimensional convolution). It can perform load/store operations to bring blocked 3-dimensional tensors from DRAM into a contiguous region of SRAM. It also provides specialized on-chip memories for network parameters, layer inputs (narrow data type), and layer outputs (wide data type). Finally, VDLA provides explicit synchronization control over successive loads, computes, and stores to maximize the overlap between memory and compute operations.

Methodology. We implemented the VDLA design on a low-power PYNQ board that incorporates an ARM Cortex A9 dual core CPU clocked at 667MHz and an Artix-7 based FPGA fabric. On these modest FPGA resources, we implemented a 16×16 matrix-vector unit clocked at 200MHz that performs products of 8-bit values and accumulates them into a 32-bit register every cycle. The theoretical peak throughput of this VDLA design is about 102.4GOPS/s. We allocated 32kB of resources for activation storage, 32kB for parameter storage, 32kB for microcode buffers, and 128kB for the register file. These on-chip buffers are by no means large enough to provide sufficient on-chip storage for a single layer of ResNet and therefore enable a case study on effective memory reuse and latency hiding.

We built a driver library for VDLA with a C runtime API that constructs instructions and pushes them to the target accelerator for execution. Our code generation algorithm then translates the accelerator program to a series of calls into the runtime API. Adding the specialized accelerator back-end took ∼2k LoC in Python.

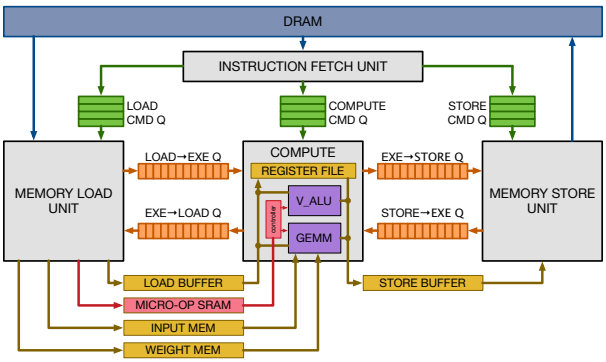


Figure 20: VDLA Hardware design overview.

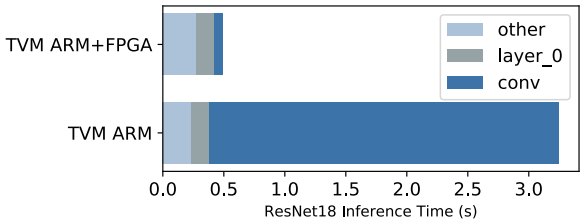


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

End-to-End ResNet Evaluation. We used TVM to generate ResNet inference kernels on the PYNQ platform and offloaded as many layers as possible to VDLA. We also used it to generate both schedules for the CPU only and CPU+FPGA implementation. Due to its shallow convolution depth, the first ResNet convolution layer could not be efficiently offloaded on the FPGA and was instead computed on the CPU. All other convolution layers in ResNet, however, were amenable to efficient offloading. Operations like residual layers and activations were also performed on the CPU since VDLA does not support these operations.

Figure 21 breaks down ResNet inference time into CPU-only execution and CPU+FPGA execution. Most computation was spent on the convolution layers that could be offloaded to VDLA. For those convolution layers, the achieved speedup was 40×. Unfortunately, due to Amdahl's law, the overall performance of the FPGA accelerated system was bottlenecked by the sections of the workload that had to be executed on the CPU. We envision that extending the VDLA design to support these other operators will help reduce cost even further. This FPGA-based experiment showcases TVM's ability to adapt to new architectures and the hardware intrinsics they expose.

7 Related Work

Deep learning frameworks [3,4,7,9] provide convenient interfaces for users to express DL workloads and deploy them easily on different hardware back-ends. While existing frameworks currently depend on vendor-specific tensor operator libraries to execute their workloads, they can leverage TVM's stack to generate optimized code for a larger number of hardware devices.

High-level computation graph DSLs are a typical way to represent and perform high-level optimizations. Tensorflow's XLA [3] and the recently introduced DLVM [45] fall into this category. The representations of computation graphs in these works are similar, and a high-level computation graph DSL is also used in this paper. While graph-level representations are a good fit for high-level optimizations, they are too high level to optimize tensor operators under a diverse set of hardware back-ends. Prior work relies on specific lowering rules to directly generate low-level LLVM or resorts to vendor-crafted libraries. These approaches require significant engineering effort for each hardware back-end and operator-variant combination.

Halide [32] introduced the idea of separating computing and scheduling. We adopt Halide's insights and reuse its existing useful scheduling primitives in our compiler. Our tensor operator scheduling is also related to other work on DSL for GPUs [18, 24, 36, 37] and polyhedral-based loop transformation [6,43]. TACO [23] introduces a generic way to generate sparse tensor operators on CPU. Weld [30] is a DSL for data processing tasks. We specifically focus on solving the new scheduling challenges of DL workloads for GPUs and specialized accelerators. Our new primitives can potentially be adopted by the optimization pipelines in these works.

High-performance libraries such as ATLAS [46] and FFTW [15] use auto-tuning to get the best performance. Tensor comprehension [42] applied black-box auto-tuning together with polyhedral optimizations to optimize CUDA kernels. OpenTuner [5] and existing hyper parameter-tuning algorithms [26] apply domain-agnostic search. A predefined cost model is used to automatically schedule image processing pipelines in Halide [29]. TVM's ML model uses effective domain-aware cost modeling that considers program structure. The based distributed schedule optimizer scales to a larger search space and can find state-of-the-art kernels on a large range of supported back-ends. More importantly, we provide an end-to-end stack that can take descriptions directly from DL frameworks and jointly optimize together with the graph-level stack.

Despite the emerging popularity of accelerators for deep learning [11, 21], it remains unclear how a compilation stack can be built to effectively target these devices. The VDLA design used in our evaluation provides a generic way to summarize the properties of TPU-like accelerators and enables a concrete case study on how to compile code for accelerators. Our approach could potentially benefit existing systems that compile deep learning to FPGA [34,40], as well. This paper provides a generic solution to effectively target accelerators via tensorization and compiler-driven latency hiding.

8 Conclusion

We proposed an end-to-end compilation stack to solve fundamental optimization challenges for deep learning across a diverse set of hardware back-ends. Our system includes automated end-to-end optimization, which is historically a labor-intensive and highly specialized task. We hope this work will encourage additional studies of end-to-end compilation approaches and open new opportunities for DL system software-hardware co-design techniques.

Acknowledgement

We would like to thank Ras Bodik, James Bornholt, Xi Wang, Tom Anderson and Qiao Zhang for their thorough feedback on earlier versions of this paper. We would also like to thank members of Sampa, SAMPL and Systems groups at the Allen School for their feedback on the work and manuscript. We would like to thank the anonymous OSDI reviewers, and our shepherd, Ranjita Bhagwan, for helpful feedbacks. This work was supported in part by a Google PhD Fellowship for Tianqi Chen, ONR award #N00014-16-1-2795, NSF under grants CCF-1518703, CNS-1614717, and CCF-1723352, and gifts from Intel (under the CAPA program), Oracle, Huawei and anonymous sources.

References

[1] NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU, 2017.

[2] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[4] AGARWAL, A., AKCHURIN, E., BASOGLU, C., CHEN, G., CYPHERS, S., DROPPO, J., EVERSOLE, A., GUENTER, B., HILLEBRAND, M., HOENS, R., HUANG, X., HUANG, Z., IVANOV, V., KAMENEV, A., KRANEN, P., KUCHAIEV, O., MANOUSEK, W., MAY, A., MITRA, B., NANO, O., NAVARRO, G., ORLOV, A., PADMILAC, M., PARTHASARATHI, H., PENG, B., REZNICHENKO, A., SEIDE, F., SELTZER, M. L., SLANEY, M., STOLCKE, A., WANG, Y., WANG, H., YAO, K., YU, D., ZHANG, Y., AND ZWEIG, G. An introduction to computational networks and the computational network toolkit. Tech. Rep. MSR-TR-2014-112, August 2014.

[5] ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques* (Edmonton, Canada, August 2014).

[6] BAGHDADI, R., BEAUGNON, U., COHEN, A., GROSSER, T., KRUSE, M., REDDY, C., VERDOOLAEGE, S., BETTS, A., DONALDSON, A. F., KETEMA, J., ABSAR, J., HAASTREGT, S. V., KRAVETS, A., LOKHMOTOV, A., DAVID, R., AND HAJIYEV, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)* (Washington, DC, USA, 2015), PACT '15, IEEE Computer Society, pp. 138–149.

[7] BASTIEN, F., LAMBLIN, P., PASCANU, R., BERGSTRA, J., GOODFELLOW, I. J., BERGERON, A., BOUCHARD, N., AND BENGIO, Y. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[8] CHEN, T., AND GUESTRIN, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, ACM, pp. 785–794.

[9] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., , AND ZHANG, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)* (2015).

[10] CHEN, T.-F., AND BAER, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44*, 5 (May 1995), 609–623.

[11] CHEN, Y., LUO, T., LIU, S., ZHANG, S., HE, L., WANG, J., LI, L., CHEN, T., XU, Z., SUN, N., AND TEMAM, O. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 609–622.

[12] CHEN, Y.-H., EMER, J., AND SZE, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 367–379.

[13] COURBARIAUX, M., BENGIO, Y., AND DAVID, J. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR abs/1511.00363* (2015).

[14] EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., STAMM, R. L., AND TULLSEN, D. M. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro 17*, 5 (Sept 1997), 12–19.

[15] FRIGO, M., AND JOHNSON, S. G. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on* (May 1998), vol. 3, pp. 1381–1384 vol.3.

[16] HE, K., ZHANG, X., REN, S., AND SUN, J. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027* (2016).

[17] HEGARTY, J., BRUNHAVER, J., DEVITO, Z., RAGAN-KELLEY, J., COHEN, N., BELL, S., VASILYEV, A., HOROWITZ, M., AND HANRAHAN, P. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph. 33*, 4 (July 2014), 144:1–144:11.

[18] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 556–571.

[19] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR abs/1704.04861* (2017).

[20] JOUPPI, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture* (May 1990), pp. 364–373.

[21] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., BOYLE, R., CANTIN, P.-L., CHAO, C., CLARK, C., CORIELL, J., DALEY, M., DAU, M., DEAN, J., GELB, B., GHAEMMAGHAMI, T. V., GOTTIPATI, R., GULLAND, W., HAGMANN, R., HO, C. R., HOGBERG, D., HU, J., HUNDT, R., HURT, D., IBARZ, J., JAFFEY, A., JAWORSKI, A., KAPLAN, A., KHAITAN, H., KILLEBREW, D., KOCH, A., KUMAR, N., LACY, S., LAUDON, J., LAW, J., LE, D., LEARY, C., LIU, Z., LUCKE, K., LUNDIN, A., MACKEAN, G., MAGGIORE, A., MAHONY, M., MILLER, K., NAGARAJAN, R., NARAYANASWAMI, R., NI, R., NIX, K., NORRIE, T., OMERNICK, M., PENUKONDA, N., PHELPS, A., ROSS, J., ROSS, M., SALEK, A., SAMADIANI, E., SEVERN, C., SIZIKOV, G., SNELHAM, M., SOUTER, J., STEINBERG, D., SWING, A., TAN, M., THORSON, G., TIAN, B., TOMA, H., TUTTLE, E., VASUDEVAN, V., WALTER, R., WANG, W., WILCOX, E., AND YOON, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 1–12.

[22] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[23] KJOLSTAD, F., KAMIL, S., CHOU, S., LUGATO, D., AND AMARASINGHE, S. The tensor algebra compiler. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 77:1–77:29.

[24] KLÖCKNER, A. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, Scotland., 2014), Association for Computing Machinery.

[25] LAVIN, A., AND GRAY, S. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* (2016), pp. 4013–4021.

[26] LI, L., JAMIESON, K. G., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR abs/1603.06560* (2016).

[27] LIU, D., CHEN, T., LIU, S., ZHOU, J., ZHOU, S., TEMAN, O., FENG, X., ZHOU, X., AND CHEN, Y. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 369–381.

[28] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[29] MULLAPUDI, R. T., ADAMS, A., SHARLET, D., RAGAN-KELLEY, J., AND FATAHALIAN, K. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph. 35*, 4 (July 2016), 83:1–83:11.

[30] PALKAR, S., THOMAS, J. J., NARAYANAN, D., SHANBHAG, A., PALAMUTTAM, R., PIRK, H., SCHWARZKOPF, M., AMARASINGHE, S. P., MADDEN, S., AND ZAHARIA, M. Weld: Rethinking the interface between data-intensive applications. *CoRR abs/1709.06416* (2017).

[31] RADFORD, A., METZ, L., AND CHINTALA, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[32] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[33] RASTEGARI, M., ORDONEZ, V., REDMON, J., AND FARHADI, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.

[34] SHARMA, H., PARK, J., MAHAJAN, D., AMARO, E., KIM, J. K., SHAO, C., MISHRA, A., AND ESMAEILZADEH, H. From high-level deep neural models to fpgas. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–12.

[35] SMITH, J. E. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1982), ISCA '82, IEEE Computer Society Press, pp. 112–119.

[36] STEUWER, M., REMMELG, T., AND DUBACH, C. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2017), CGO '17, IEEE Press, pp. 74–85.

[37] SUJEETH, A. K., LEE, H., BROWN, K. J., CHAFI, H., WU, M., ATREYA, A. R., OLUKOTUN, K., ROMPF, T., AND ODERSKY, M. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (USA, 2011), ICML'11, pp. 609–616.

[38] TAI, K. S., SOCHER, R., AND MANNING, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[39] TULLOCH, A., AND JIA, Y. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427* (2017).

[40] UMUROGLU, Y., FRASER, N. J., GAMBARDELLA, G., BLOTT, M., LEONG, P. H. W., JAHRE, M., AND VISSERS, K. A. FINN: A framework for fast, scalable binarized neural network inference. *CoRR abs/1612.07119* (2016).

[41] VASILACHE, N. personal communication.

[42] VASILACHE, N., ZINENKO, O., THEODORIDIS, T., GOYAL, P., DEVITO, Z., MOSES, W. S., VERDOOLAEGE, S., ADAMS, A., AND COHEN, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR abs/1802.04730* (2018).

[43] VERDOOLAEGE, S., CARLOS JUEGA, J., COHEN, A., IGNACIO GÓMEZ, J., TENLLADO, C., AND CATTHOOR, F. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan. 2013), 54:1–54:23.

[44] VOLKOV, V. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California at Berkeley, 2016.

[45] WEI, R., ADVE, V., AND SCHWARTZ, L. Dlvm: A modern compiler infrastructure for deep learning systems. *CoRR abs/1711.03016* (2017).

[46] WHALEY, R. C., AND DONGARRA, J. J. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1998), SC '98, IEEE Computer Society, pp. 1–27.

[47] WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[48] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

Gandiva: Introspective Cluster Scheduling for Deep Learning

Wencong Xiao[†⋆*], Romil Bhardwaj[⋆*], Ramachandran Ramjee[⋆], Muthian Sivathanu[⋆], Nipun Kwatra[⋆], Zhenhua Han[◇⋆], Pratyush Patel[⋆], Xuan Peng[‡⋆], Hanyu Zhao[§⋆], Quanlu Zhang[⋆], Fan Yang[⋆], Lidong Zhou[⋆]
[†]*Beihang University,* [⋆]*Microsoft Research,* [◇]*The University of Hong Kong,*
[‡]*Huazhong University of Science and Technology,* [§]*Peking University*

Abstract

We introduce Gandiva, a new cluster scheduling framework that utilizes domain-specific knowledge to improve latency and efficiency of training deep learning models in a GPU cluster.

One key characteristic of deep learning is feedback-driven exploration*, where a user often runs a set of jobs (or a multi-job) to achieve the best result for a specific mission and uses early feedback on accuracy to dynamically prioritize or kill a subset of jobs; simultaneous early feedback on the entire multi-job is critical. A second characteristic is the heterogeneity of deep learning jobs in terms of resource usage, making it hard to achieve best-fit a priori. Gandiva addresses these two challenges by exploiting a third key characteristic of deep learning:* intra-job predictability*, as they perform numerous repetitive iterations called mini-batch iterations. Gandiva exploits intra-job predictability to time-slice GPUs efficiently across multiple jobs, thereby delivering low-latency. This predictability is also used for introspecting job performance and dynamically migrating jobs to better-fit GPUs, thereby improving cluster efficiency.*

We show via a prototype implementation and micro-benchmarks that Gandiva can speed up hyper-parameter searches during deep learning by up to an order of magnitude, and achieves better utilization by transparently migrating and time-slicing jobs to achieve better job-to-resource fit. We also show that, in a real workload of jobs running in a 180-GPU cluster, Gandiva improves aggregate cluster utilization by 26%, pointing to a new way of managing large GPU clusters for deep learning.

1 Introduction

All ~~men~~ schedulers make mistakes; only the wise learn from their mistakes.

-Winston Churchill

An increasingly popular computing trend over the last few years is deep learning [32]; it has already had significant impact; *e.g.*, on widely-used personal products for voice and image recognition, and has significant potential to impact businesses. Hence, it is likely to be a vital and growing workload, especially in cloud data centers.

However, deep learning is compute-intensive and hence heavily reliant on powerful but expensive GPUs; a GPU VM in the cloud costs nearly 10x that of a regular VM. Cloud operators and large companies that manage clusters of tens of thousands of GPUs rely on cluster schedulers to ensure efficient utilization of the GPUs.

Despite the importance of efficient scheduling of deep learning training (DLT) jobs, the common practice today [12, 28] is to use a traditional cluster scheduler, such as Kubernetes [14] or YARN [50], designed for handling big-data jobs such as MapReduce [17]; a DLT job is treated simply as yet another big-data job that is allocated a set of GPUs at job startup and holds exclusive access to its GPUs until completion.

In this paper, we present *Gandiva*, a new scheduling framework that demonstrates that a significant increase in cluster efficiency can be achieved by *tailoring* the scheduling framework to the unique characteristics of the deep learning workload.

One key characteristic of DLT jobs is *feedback-driven exploration* (Section 2). Because of the inherent trial-and-error methodology of deep learning experimentation, users typically try several configurations of a job (a *multi-job*), and use early feedback from these jobs to decide whether to prioritize or kill some subset of them. Such conditional exploration, called hyper-parameter search, can either be manual or automated [10, 33, 41]. Traditional schedulers run a subset of jobs to completion while queueing others; this model is a misfit for multi-jobs, which require simultaneous early feedback on all jobs within the multi-job. Also, along with multi-jobs, other DLT jobs that have identified the right hyper-parameters, run for several hours to days, leading to

[*]The first two authors have equal contribution. This work is done while Wencong Xiao, Zhenhua Han, Xuan Peng, and Hanyu Zhao are interns in Microsoft Research.

head-of-line-blocking, as long-running jobs hold exclusive access to the GPUs until completion, while multi-jobs depending on early feedback wait in queue. Long queueing times force users to either use reserved GPUs, or demand cluster over-provisioning, thus reducing cluster efficiency.

Second, like any other cluster workload, DLT jobs are heterogeneous because of the diverse application domains they target. Jobs widely differ in terms of memory usage, GPU core utilization, sensitivity to interconnect bandwidth, and/or interference from other jobs. For example, certain multi-GPU DLT jobs may perform much better with affinitized GPUs, while other jobs may not be as sensitive to affinity (Section 3). A traditional scheduler that treats a job as a black-box will hence achieve sub-optimal cluster efficiency.

To address the twin problems of high latency and low efficiency, *Gandiva* exploits a powerful property of DLT jobs: *intra-job predictability* (Section 3). A job is comprised of millions of similar, clearly separated mini-batch iterations. For example, the GPU RAM usage of a DLT job follows a cyclic pattern aligned with mini-batch boundaries, usually with more than 10x difference in GPU RAM usage within a mini-batch. *Gandiva* exploits this cyclic predictability to implement efficient *application aware time-slicing*; in effect, it re-defines the atom of scheduling from a *job* to automatically-partitioned *micro-tasks*. This enables the cluster to over-subscribe DLT jobs and provide *early feedback* through time-slicing to all DLT jobs, including all jobs that are part of a multi-job.

Gandiva also uses the predictability to perform *profile-driven introspection*. It uses the mini-batch progress rate to introspect its decisions continuously to improve cluster efficiency (Section 4). For example, it packs multiple jobs on the same GPU only when they have low memory and GPU utilization; it dynamically migrates a communication intensive job to more affinitized GPUs; it also opportunistically "grows" the degree of parallelism of a job to make use of spare resources, and shrinks the job when the spare resources go away. The introspection policy we presently implement is a stateful trial-and-error policy that is feasible because of the predictability and the limited state space of options we consider.

Beyond the specific introspection and scheduling policy evaluated in this paper, the *Gandiva* framework provides the following APIs that any DLT scheduling policy can leverage: (a) efficient suspend-resume or time-slicing, (b) low-latency migration, (c) fine-grained profiling, (d) dynamic intra-job elasticity, and (e) dynamic prioritization. The key to making these primitives efficient and practical is the co-design approach of *Gandiva* that spans across both the scheduler layer and the DLT toolkit layer such as Tensorflow [8] or PyTorch [38]. Traditional schedulers, for a good reason, treat a job as a black-box. However, by exploiting the dedicated nature of GPU clusters, *Gandiva* customizes the scheduler to the specific workload of deep learning, thus providing the scheduler more visibility and control into a job, while still achieving generality to arbitrary DLT jobs.

We have implemented *Gandiva* by modifying two popular frameworks, PyTorch and Tensorflow, to provide the necessary new primitives to the scheduler, and also implemented an initial scheduling policy manager on top of Kubernetes and Docker containers (Section 5). We evaluate *Gandiva* on a cluster of 180 heterogeneous GPUs and show, through micro-benchmarks and real workloads, that (i) *Gandiva* improves the efficiency of cluster scheduling by up to 26%, and (ii) *Gandiva* is reactive enough to time-slice multiple jobs dynamically on the same GPU, reducing the time to early feedback by as much as 77%. We also show that, for a popular hyper-parameter search technique [10], *Gandiva* improves the overall completion time of the hyper-parameter search by up to an order of magnitude while using same resources (Section 6).

The key contributions of the paper are as follows.

- We illustrate various unique characteristics of the deep learning workflow and map it to specific requirements needed for cluster scheduling.
- We identify generic primitives that can be used by a DLT job scheduling policy, and provide application-aware techniques to make primitives such as time-slicing and migration an order of magnitude more efficient and thus practical by leveraging DL-specific knowledge of intra-job periodicity.
- We propose and evaluate a new introspective scheduling framework that utilizes domain-specific knowledge of DLT jobs to refine its scheduling decision continuously, thereby significantly improving early feedback time and delivering high cluster efficiency.

2 Background

Deep learning is a type of representation learning that automatically infers features from raw data in order to accomplish tasks such as image classification or language translation [32]. Deep learning may be supervised (data with labels) or unsupervised (data only). In either case, the representation is a *deep neural network model* with parameters called weights. These weights are carefully arranged in layers and number typically in the millions. These model weights are learned through *training*.

Deep learning training operates on a few samples of data at a time called a *mini-batch*. It computes a set of scores for each mini-batch by performing numerical

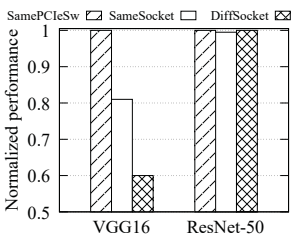


Figure 1: Intra-server locality.

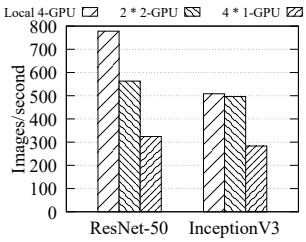


Figure 2: Inter-server locality.

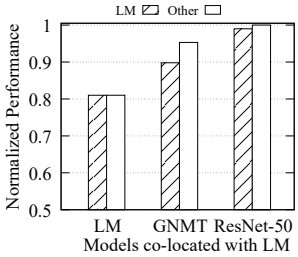


Figure 3: 1-GPU interference.

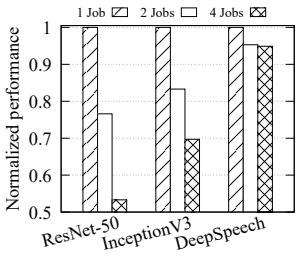


Figure 4: NIC interference.

computations using the model weights, called the *forward pass*. Based on the desired task, an objective function is defined that measures an error between the computed scores and desired scores. The error is populated via a *backward pass* over the model, where it first computes a gradient for each weight (i.e., the impact of each weight on the error) and then applies a negative of the gradient, scaled by a parameter called the learning rate, to each weight to decrease the error. Both the forward and backward passes typically involve billions of floating point operations and thus leverage GPUs. Each forward-backward pass is called a mini-batch iteration. Typically, millions of such iterations are performed on large datasets to achieve high task accuracy.

Feedback-driven exploration. One pre-requisite for achieving high accuracy is model selection. Discovery of new models such as ResNet [24] or Inception [46] is mostly a trial-and-error process today, though ways to automate it is an active area of research [36].

Apart from the model structure, there are a number of parameters, called *hyper-parameters*, that also need to be specified as part of the DLT job. Hyper-parameters include the number of layers/weights in the model, mini-batch size, learning rate, etc. These are typically chosen today by the user based on domain knowledge and trial-and-error, and can sometimes even result in early training failure. Thus, *early-feedback* on DLT jobs is critical, especially in the initial stages of training.

Multi-job. Once the user has identified a particular model to explore further, the user typically performs *hyper-parameter search* to improve task accuracy. This can be done using various searching techniques over the space of the hyper-parameters; that is, the user generates *multiple DLT jobs* or multi-jobs, each performing full training using one set of hyper-parameters or configuration. Because users typically explore hundreds of such configurations, this process is computationally expensive. Thus, sophisticated versions of hyper-parameter searches are available in the literature, such as HyperOpt [10] and Hyperband [33]. For example, Hyperband might initially spawn 128 DLT jobs and, in each round (*e.g.*, 100 mini-batch iterations), kill half of the jobs with the lowest accuracy. Again, for these algorithms, *early feedback* on the entire set of jobs is crucial because they would be unable to make effective training decisions otherwise.

3 DLT Job Characteristics

In this section, we motivate the design of *Gandiva* by highlighting several unique characteristics of DLT jobs.

3.1 Sensitivity to locality

The performance of a multi-GPU DLT job depends on the affinity of the allocated GPUs. Different DLT jobs exhibit different levels of sensitivity to inter-GPU affinity. Even for GPUs on the same machine, we observe different levels of inter-GPU affinity due to asymmetric architecture: two GPUs might be located in different CPU sockets (denoted as DiffSocket), in the same CPU socket, but on different PCIe switches (denoted as SameSocket), or on the same PCIe switch (denoted as SamePCIeSw).

Figure 1 shows different sensitivity to *intra-server locality* for two models VGG16 [44] and ResNet-50 [24]. When trained with two P100 GPUs using Tensorflow, VGG16 suffers greatly under bad locality. With the worst locality, when two GPUs are located in different CPU sockets, VGG16 achieves only 60% of the best locality config, where two GPUs are placed under the same PCIe switch. On the other hand, the ResNet-50 is not affected by GPU locality in this setting. This is because VGG16 is a larger neural model than ResNet-50, hence the model synchronization in each mini-batch incurs a higher communication load on the underlying PCIe bus.

We observe similar trends in a distributed setting. Figure 2 shows the performance of a 4-GPU Tensorflow job running with different *inter-server locality*, training ResNet-50 and InceptionV3 [46] models. Even when interconnected with a 40G InfiniBand network, the performance difference is clearly seen when the job is assigned to 4 GPUs, where they are evenly scattered across 4 servers (denoted as 4*1-GPU), 2 servers (denoted as 2*2-GPU), and all in one server (denoted as local 4-GPU), though the sensitivity to locality of the two models is different.

Thus, a DLT scheduler has to take into account a job's sensitivity to locality when allocating GPUs.

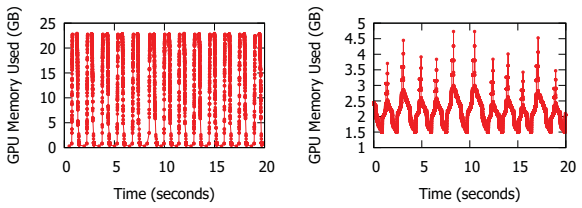
3.2 Sensitivity to interference

When running in a shared execution environment, DLT jobs might interfere with each other due to resource contention. We again observe that different DLT jobs exhibit different degrees of interference.

Interference exists even for single-GPU jobs. When placing a Language Model [56] job (marked as LM) with another job under the same PCI-e switch, Figure 3 shows the performance degradation due to *intra-server* interference. When two LMs run together, both jobs suffer 19% slowdown. However, ResNet-50 does not suffer from GPU co-location with LM. Neural Machine Translation (GNMT) [51] exhibits a modest degree of interference with LM. Similarly, we also observe various degrees of interference for multi-GPU training with different types of training models. We omit the result due to space limitation.

Figure 4 shows *inter-server interference* on two 4-GPU servers that are connected with a 40G InfiniBand network. When running multiple 2-GPU jobs, where each GPU is placed on different server, ResNet-50 shows up to 47% slowdown, InceptionV3 shows 30% slowdown, while DeepSpeech [23] only shows 5% slowdown.

In summary, popular deep learning models across different application domains such as vision, language, and speech demonstrate different levels of sensitivity to locality and interference. To cater to these challenges, *Gandiva* leverages a key characteristic of DLT jobs, which we elaborate next.



(a) ResNet50/Imagenet (b) GNMT/WMT'14 En-De

Figure 5: GPU memory usage during training.

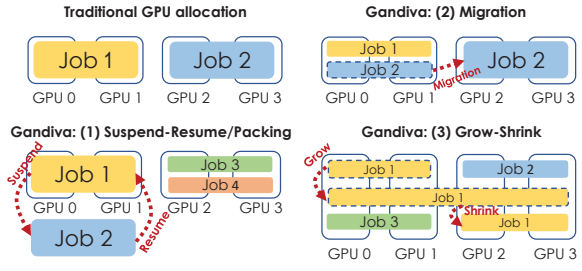


Figure 6: GPU usage options in Gandiva.

3.3 Intra-job predictability

A DLT job consists of numerous mini-batch iterations. The total GPU memory used[1] during a 20s snapshot of training on ImageNet data when using ResNet-50 model [24] on four K80 GPUs is shown in Figure 5(a). The GPU memory used clearly follows a cyclic pattern. Each of these cycles corresponds to the processing of a single mini-batch (about 1.5s), with the memory increasing during the forward pass and decreasing during the backward pass. The maximum and minimum GPU memory used is 23GB and 0.3GB, respectively, or a factor of 77x. This ratio scales with the mini-batch size (typically between 16 to 256; 128 in this case).

The total GPU memory used during a 20s snapshot of training on WMT'14 English German language dataset when using GNMT model [51] on one K80 GPU is shown in Figure 5(b). While the mini-batch iterations are not identical to each other as in the ImageNet example (due to differing sentence lengths and the use of dynamic graphs in PyTorch), the graph has a similar cyclic nature. The difference between maximum and minimum is smaller (3x) primarily due to larger model (0.4GB) and smaller mini-batch size (16 in this example).

Apart from image and language models shown here, other training domains such as speech, generative adverserial networks (GANs), and variational auto-encoders all follow a similar cyclic pattern (not shown due to space limitation) since the core of training is the gradient descent algorithm performing many mini-batch iterations.

Leveraging predictability. This characteristic behavior is exploited in *Gandiva* in multiple ways. First, a DLT job can be automatically split into mini-batch iterations and a collection of these iterations over 60 seconds, say a micro-task, forms a scheduling interval. Second, by performing the suspend operation at the minimum of the memory cycle, the amount of memory to be copied from GPU to be saved in CPU can be significantly reduced, thereby enabling suspend/resume and migration to be an order of magnitude more efficient than a naïve implementation. Third, the mini-batch progress rate can be profiled and used as a proxy to evaluate the effectiveness of applying mechanisms such as packing or migration.

4 Design

High latency and low utilization in today's cluster arises because DLT jobs are assigned a *fixed set* of GPUs *exclusively* (Figure 6). Exclusive access to GPUs causes

[1]This is actual GPU memory used. Toolkits like PyTorch/Tensorflow use caching to avoid expensive GPU memory (de)allocations.

head-of-line blocking, preventing early feedback and resulting in high queuing times for incoming jobs. Exclusive access to a fixed set of GPUs also results in low GPU utilization when jobs are unable to utilize their assigned GPUs fully.

4.1 Mechanisms

In *Gandiva*, we address these inefficiencies by removing the exclusivity and fixed assignment of GPUs to DLT jobs in three ways (Figure 6). First, during overload, instead of waiting for current jobs to depart, *Gandiva* allows incoming jobs to time-share GPUs with existing jobs. This is enabled using a custom suspend-resume mechanism tailored for DLT jobs along with selective packing. Second, *Gandiva* supports efficient migration of DLT jobs from one set of GPUs to another. Migration allows time-sliced jobs to migrate to other (recently vacated) GPUs or for de-fragmentation of the cluster so that incoming jobs are assigned GPUs with good locality. Third, *Gandiva* supports a GPU grow-shrink mechanism so that idle GPUs can be used opportunistically. In order to support these mechanisms efficiently and enable effective resource management, *Gandiva* introspects DLT jobs by continuously profiling their resource usage and estimating their performance. We now describe each of these mechanisms.

Suspend-Resume and Packing. Suspend-resume is one mechanism *Gandiva* uses to *remove exclusivity* of a set of GPUs to a DLT job. Modern operating systems support efficient suspend-resume for CPU process time-slicing. *Gandiva* leverages this mechanism and adds custom support for GPU time-slicing.

As shown in Figure 5(a), usage of GPU memory by DLT jobs has a cyclic pattern with as much as 77x difference between the minimum and maximum memory usage. The key idea in *Gandiva* is to exploit this cyclic behavior and suspend-resume DLT jobs when their GPU memory usage is at their lowest. Thus, when a suspend call is issued, the DLT toolkit waits until the minimum of the memory usage cycle, copies the objects stored in the GPU to the CPU, releases all its GPU memory allocations (including cache), and then invokes the classic CPU suspend mechanism. Later, when the CPU resumes the job, the DLT framework first allocates appropriate GPU memory, copies the stored objects back to the GPU, and then resumes the job.

Suspend-resume may also initiate a change of GPU within the same server (*e.g.*, in the case of six 1-GPU jobs time-sharing 4-GPUs). While changing GPU is expensive, we hide this latency from the critical path. As we show in our evaluation (Section 6.1), for typical image classification jobs, suspend-resume together can be accomplished in under 100ms, while for large language translation jobs suspend-resume can take up to 1s. Given a time-slicing interval of 1 minute, this amounts to an overhead of 2% or less.

Note that suspend in *Gandiva* may be delayed by at most a mini-batch interval of the DLT job (typically, a few seconds or less), but we believe this is a worthwhile trade-off as it results in significantly less overhead due to the reduced GPU-CPU copy cost and less memory used in the CPU. Further, useful work is accomplished during this delay. The scheduler keeps track of this delay and adjusts the time-slicing interval accordingly for fairness.

An alternative to suspend-resume for time-slicing is to run multiple DLT jobs on a GPU *simultaneously* and let the GPU time-share the jobs. We call this *packing*. Packing in GPU is efficient only when the packed jobs do not exceed the GPU resources (cores, memory) and do not adversely impact each other. If jobs interfere, packing can be significantly worse than suspend-resume (Section 6.1). We use profiling to monitor the resource and progress of DLT jobs when they have exclusive access. If two jobs are identified as candidates for packing, we pack them together and continue monitoring them. If a given packing results in adverse impact on jobs' performance, we unpack those jobs and revert to suspend-resume.

Migration. Migration is the mechanism *Gandiva* uses to *change the set* of GPUs assigned to a DLT job. Migration is useful in several situations such as i) moving time-sliced jobs to vacated GPUs anywhere in the cluster; ii) migrating interfering jobs away from each other; iii) de-fragmentation of the cluster so that incoming jobs get GPUs with good locality.

We evaluate two approaches for tackling DLT process state migration. In the first approach, we leverage a generic process migration mechanism such as CRIU [1]. Because CRIU by itself does not support migration of processes that use the GPU device, we first checkpoint GPU objects and remove all GPU state from the process before CRIU is invoked. Because CRIU checkpoints and restores the entire process memory, the size of the checkpoint is on the order of GBs for these DLT jobs using PyTorch. Thus, the resulting migration overhead is about 8-10s for single GPU jobs and higher for multi-GPU jobs.

The second approach we consider is the use of DLT jobs that are checkpoint-aware. DLT frameworks such as Tensorflow already support APIs (*e.g.*, `tensorflow.train.saver`) that allow automatic checkpoint and restore of models. This API is used today to ensure that long running jobs do not have to be rerun due to server failures. We extend the framework to support migration of such jobs. By *warming up* the destination before migration and only migrating the necessary training state, we can reduce the migration over-

head to as little as a second or two (Section 6.1). With either approach, we find that the overhead of inter-server migration is worthwhile compared to the benefits it provides in terms of higher overall GPU utilization.

Grow-Shrink. The third mechanism that *Gandiva* uses to remove the exclusivity of GPUs to a DLT job is grow-shrink. This mechanism primarily targets situations when the cluster may not be fully utilized, say, late at night. The basic idea is to grow the number of GPUs available to a job opportunistically during idle times and correspondingly also shrink the number of GPUs available when the load increases.

Many DLT jobs, especially in the image domain, see linear performance scaling as the number of GPUs is increased. *Gandiva* applies this mechanism only to those DLT jobs that specifically declare that they are adaptive enough to take advantage of these growth opportunities. When multiple DLT jobs fit this criteria, *Gandiva* uses profiling information, discussed next, to estimate each job's progress rate and then allocate GPUs accordingly.

Profiling. Like any scheduler, *Gandiva* monitors resource usage such as CPU and GPU utilization, CPU/GPU memory, etc. However, what is unique to *Gandiva* is that it also *introspects* DLT jobs in an application-aware manner to estimate their rate of progress. This introspection exploits the regular pattern exhibited by DLT jobs (Section 3) and uses the periodicity to estimate their progress rate.

Gandiva estimates a DLT job's `mini_batch_time`, the time to do one forward/backward pass over a batch of input data, as the time taken between two minimums of the GPU memory usage cycles (Figure 5(a)). Because DLT jobs typically perform millions of such mini batch operations in their lifetime, the scheduler compares the `mini_batch_time` of a DLT prior to and post a scheduling decision to determine its effectiveness.

For example, consider the example of packing two DLT jobs in a GPU described earlier. By comparing the `mini_batch_time` of each of the two DLT jobs before and after packing, *Gandiva* can decide whether packing is effective. Without such profiling, in order to make a packing decision, one would have to model not only the two DLT jobs' performance on various GPUs but also the various ways in which they may interfere with each other (*e.g.*, caches, memory bandwidth, etc.), a non-trivial task as evidenced by the varied performance of packing we see in Section 6.1.

4.2 Scheduling Policy

Definitions: Before we describe the details of the scheduler, we define some terminology. DLT jobs are encapsulated in containers (Section 5) and include the number of GPUs required, their priority (can be dynamically changed), and a flag indicating if the job is capable of grow-shrink. We assume the number of GPUs requested by a job is a power of two (typical for DLT jobs today). A cluster is composed of one or more servers, with each server having one or more GPUs. Further, we assume a dedicated GPU cluster for DLT jobs [28, 12].

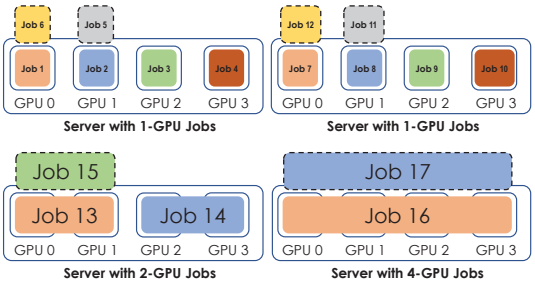


Figure 7: Scheduling example in a 16-GPU Cluster.

We define the *height* of a server as $\lceil M/N \rceil$, where M is the number of allocated GPUs and N is the number of total GPUs. Thus, the suspend/resume mechanism will only be used when the height of a server exceeds one. The height of a cluster is defined as the maximum height of all its servers. *Overload* occurs when the height of the cluster is greater than one; i.e., the sum of requested/allocated GPUs of all jobs is greater than the total number of GPUs. We define the *affinity* of a server as the type of jobs (based on GPUs required) assigned to that server. For example, initially servers have affinity of zero and, if a job that requires two GPUs is assigned to a server, the affinity of that server is changed to two. This parameter is used by the scheduler to assign jobs with similar GPU requirements to the same server.

Goals: The primary design goal of the *Gandiva* scheduler is to provide ***early feedback*** to jobs. In prevalent schedulers, jobs wait in a queue during overload. In contrast, *Gandiva* supports over-subscription by allocating GPUs to a new job immediately and using the suspend-resume mechanism to provide early results. A second design goal is ***cluster efficiency***. This is achieved through a continuous optimization process that uses profiling and a greedy heuristic that takes advantage of mechanisms such as packing, migration, and grow-shrink. ***Cluster-level fairness*** is not a design goal in *Gandiva*. While we believe achieving long-term fairness at the cluster level is feasible using the *Gandiva* mechanisms, in this paper, we focus only on providing fairness among jobs at each server using the suspend-resume mechanism and leave cluster-level fairness to future work.

To achieve these goals, the *Gandiva* scheduler operates in two modes: *reactive and introspective*. By reactive mode, we refer to when the scheduler reacts to events such as job arrivals, departures, machine failures etc. By introspective mode, we refer to a continuous process where the scheduler aims to improve cluster utiliza-

Algorithm 1 getNodes(in job, out nodes)

```
1:  nodes0 ← findNodes(job.gpu, affinity ← job.gpu)
2:  nodes1 ← minLoadNodes(node0)
3:  nodes2 ← findNodes(jog.gpu, affinity ← 0)
4:  nodes3 ← findNodes(job.gpu)
5:  if nodes1 and height(nodes1) < 1:
6:      return nodes1            // Same affinity with free GPUs
7:  if nodes2 and numGPUs(nodes2) ≥ job.gpu:
8:      return nodes2                // Unallocated GPU servers
9:  if nodes3:
10:     return nodes3                // Relax affinity constraint
11: elif nodes1:
12:     return nodes1                // Allow over-subscription
13: else:
14:     enqueue(job)                                  // Job queued
```

tion and job completion time. Note that the scheduler can be operating in both modes at the same time. We discuss each of these modes next.

4.2.1 Reactive Mode

The reactive mode is designed to take care of events such as job arrivals, departures, and machine failures. Conventional schedulers operate in this mode. Here we discuss only our job placement policy since we follow the conventional approach for failure handling.

When a new job arrives, the scheduler allocates servers/GPUs for the job. The node allocation policy used in *Gandiva* is shown in Algorithm 1. *findNodes* is a function to return the node candidates that satisfy the job request with an optional parameter for affinity constraint. Initially, *Gandiva* tries to find nodes with the same affinity as the new job and, among those, ones with the minimum loads. If such nodes exist and their height is less than one (lines 5–6), that node is assigned. Otherwise, *Gandiva* tries to find and assign un-affinitized nodes (lines 7–8). If no such free servers are available, the third option is to look for nodes with free GPUs while ignoring affinity (lines 9–10). This may result in fragmented allocation across multiple nodes but, as we shall see later, migration can be used for defragmentation. If none of the above work, it implies that no free GPUs are available in the cluster. In this case, if nodes with the same affinity exist, they are used with suspend-resume (lines 11–12); if not, the job is queued (lines 13–14).

For example, as shown in Figure 7, jobs that require 1-GPU are placed together but jobs that require 2 or 4 GPUs are placed on different servers. Further, we try to balance the over-subscription load on each of the servers by choosing the server with the minimum load (*e.g.*, six 1-GPU jobs on each of the two servers in the figure).

Conventional schedulers will use job departures to pick the next job from the waiting queue for placement.

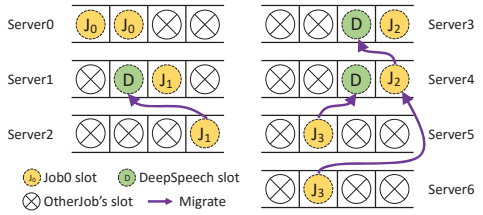


Figure 8: Job migration in a shared cluster.

In addition, in *Gandiva*, we check whether the height of the cluster can be reduced; e.g., by migrating a job that is suspended to the newly vacated GPU. This job could be from the same server or from any other server in the cluster. Finally, job departures can also trigger migrations for improving locality, as discussed in the next section.

Gandiva's job placement policy takes into account two factors. First, unlike conventional schedulers, *Gandiva* allows over-subscription. When a server is over-subscribed, we do weighted round-robin scheduling to give each job its fair time-share. Second, unlike today's schedulers, where GPU allocation is a one-time event at job arrival, *Gandiva* uses the introspective mode, discussed next, to improve cluster utilization continuously. Thus, *Gandiva* relies on a simple job placement policy to allocate GPU resources quickly to new jobs, thereby enabling early feedback.

4.2.2 Introspective Mode

In the introspective mode, *Gandiva* continuously monitors and optimizes placement of jobs to GPUs in the cluster to improve the overall utilization and the completion time of DLT jobs.

Packing. Packing is considered only during overload. The basic idea behind packing is to run two or more jobs simultaneously on a GPU to increase efficiency. If the memory requirements of the packing jobs combined are higher than GPU memory, the overhead of "paging" from CPU memory is significantly high [16] that packing is not effective. When the memory requirements of two or more jobs are smaller than GPU memory, packing still may not be more efficient than suspend-resume as we show in Section 6.1. For example, for some DLT jobs, packing increases efficiency, while for others packing can be worse than suspend-resume.

Analytically modeling performance of packing is a challenging problem given the heterogeneity of DLT jobs. Instead, *Gandiva* relies on a greedy heuristic to pack jobs. When jobs arrive, we always run them in exclusive mode using suspend-resume and collect profiling information (GPU utilization, memory and job progress rate). Based on the profiling data, the scheduler maintains a list of jobs sorted by their GPU utilization. The scheduler greedily picks the job with the lowest GPU uti-

lization and attempts to pack it on a GPU with the lowest GPU utilization. We only do this when the combined memory utilization of the packed jobs do not exceed the overall memory of the GPU. Packing is deemed successful when the total throughput of packed jobs is greater than time-slicing. If packing is unsuccessful, we undo the packing and try the next lowest utilization GPU. If the packing is successful, we find the next lower utilization job and repeat this process. Based on our evaluation, we find that this simple greedy heuristic achieves 26% efficiency gains.

Migration. GPU locality can play a significant role in the performance of some jobs (Section 3.1). In *Gandiva*, we use migration to improve locality whenever a job departs and also as a background process to "defrag" the cluster. To improve locality, we pick jobs that are not co-located and try to find a new co-located placement. Figure 8 illustrates an example from a cluster experiment (Section 6.4). When a multi-job with 4 jobs that requires 2-GPUs each was scheduled, it had poor GPU affinity; only J_0's two GPUs are colocated with the other 3 jobs in the multi-job (J_1, J_2, and J_3,) assigned to separated GPUs. Three minutes later, a background training job, DeepSpeech, completes and releases its 8 GPUs. Three of the 8 GPUs, marked as D in Figure 8 in three different servers (server 1, 3, and 4), can improve the training efficiency of the multi-job. *Gandiva* hence initiates the migration process, relocating J_1, J_2, and J_3 to colocated GPUs. For de-fragmentation, we pick the server with the most free GPUs among all non-idle ones. We then try to move the jobs running on that server to others. The job will be migrated to another server with fewer free GPUs, as long as there is negligible performance loss. We repeat this until the number of free GPUs on every non-idle server is less than a threshold (3 out of 4 in our experiments) or if no job will benefit from migration.

Grow-shrink. Grow-shrink is only triggered when the cluster is under-utilized and the DLT jobs specifically identify themselves as amenable to grow-shrink. In our current system, we only grow jobs to use up to the maximum number of GPUs available in a single server. Further, we trigger growth only after an idle period to avoid thrashing and shrink immediately when a new job might require the GPUs.

Time-slicing. Finally, we support round robin scheduling in each server to time-share GPUs fairly (Section 6.1). When jobs have multiple priority levels, higher priority jobs will never be suspended to accommodate lower priority jobs. If a server is fully utilized with higher priority jobs, the lower priority job will be migrated to another server, if feasible.

5 Implementation

DLT jobs are encapsulated as Docker containers containing our customized versions of DL toolkits and a *Gandiva* client. These jobs are submitted to a Kubernetes [14] system. *Gandiva* also implements a custom scheduler that then schedules these jobs.

5.1 Scheduler

Gandiva consists of a custom central scheduler and also a client component that is part of every DLT job container. The scheduler is just another container managed by Kubernetes. Kubernetes is responsible for overall cluster management, while the *Gandiva* scheduler manages the scheduling of DLT jobs. The *Gandiva* scheduler uses the Kubernetes API to get cluster node and container information and, whenever a new container is submitted, the scheduler assigns it to one or more of the GPUs in the cluster based on the scheduling policy.

When a container is scheduled on a node, initially only the *Gandiva* client starts executing. It then polls the *Gandiva* scheduler to identify which GPUs to make available for the DLT job and also controls the execution of the DLT job using suspend/resume and migrate commands. While scheduling of all the GPUs in our cluster is fully controlled by the central scheduler, a hierarchical approach may be needed if scalability becomes a concern.

5.2 Modifications to DL toolkits

In the interest of space, we describe only the time-slicing implementation for PyTorch and the migration implementation for Tensorflow.

PyTorch time-slicing. The *Gandiva*client issues a SIGTSTP signal to indicate that the toolkit must suspend the process. It also indicates whether or not the resume should occur in a new GPU via an in-memory file. Upon receiving the signal, the toolkit sets a suspend flag and executes the suspend only at the end of a mini-batch boundary.

In Tensorflow, a define-and-run toolkit, the mini-batch boundaries are easily identified (end of `session.run()`). In PyTorch, a define-by-run toolkit, we identify the mini-batch boundary by tracking GPU memory usage cycles as part of PyTorch's GPU memory manager (THCCachingAllocator) and looking for a cycle minimum whenever GPU memory is freed.

Once the minimum is detected, the toolkit i) copies all stored objects from GPU to CPU, ii) frees up GPU allocations, and iii) suspends the process. When *Gandiva* client issues a SIGCONT signal, the toolkit allocates GPU memory, copies stored objects from CPU to GPU, and resumes the process. To handle device address

change on resume, we track GPU objects in the toolkit and *patch* them with the new addresses. Changing GPU involves calling cudaDeviceReset and CudaInit, which can take 5-10s. We hide this latency by performing these actions in the background while "suspended".

Tensorflow migration. We make changes to Tensorflow (TF) with 400+ lines of Python/C++ code. With 200+ line of additional code, we deploy a *Migration Helper* on each server to support on-demand checkpointing and migration. When receiving a migration command from the scheduler, the destination Helper first warms up the TF session and waits for the checkpoint. The source Helper then asks TF to save the checkpoint, moves the checkpoint to destination in case of cross-server migration, and finally resumes the training session. To speed up the migration process, we adopt Ramdisk to keep the checkpoint in memory. In the cross-server case, the modified TF saves the checkpoint to the remote Ramdisk directly through the Network File System (NFS) protocol.

When the Migration Helper asks a job to perform checkpointing, the modified TF calls `tf.Saver` at the end of a mini-batch. For data parallelism, the checkpoint only includes the model in one GPU, regardless of the number of GPUs used in the training. To speedup TF migration further, we do not include the meta-graph structure in a checkpoint as it can be reconstructed based on user code.

In the warm-up phase, the modified TF checks the GPU configuration and reconstructs the meta-graph. It further creates the Executor to run a warm-up operation to ensure that the initialization is not deferred lazily. When resuming the training process, the modified TF loads the checkpoint, with multiple GPUs loading it in parallel, and continues the training.

6 Evaluation

In this section, we first present micro-benchmark results of the *Gandiva* mechanisms. We then evaluate the benefit *Gandiva* provides to multi-jobs. Finally, we present our evaluation results of the experiments on a 180-GPU cluster.

Our servers are 12-core Intel Xeon E5-2690@2.60GHz with 448GB RAM and two 40Gbps links (no RDMA), running Ubuntu 16.04. Each server has either four P100 or four P40 GPUs. All servers are connected to a network file-system called GlusterFS [3] with two-way replication on the server disks (SSDs). For jobs that use more than one GPU, we only evaluate data parallelism (as it is more common than model parallelism), and use synchronous updates (though we can support asynchronous update as well). Our evaluation uses 18 models, 8 implemented in PyTorch 0.3 and 10 implemented in TensorFlow 1.4. The batch size

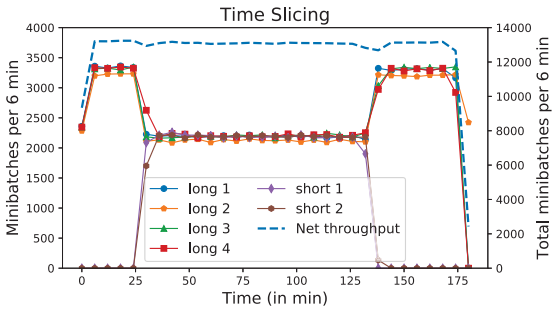


Figure 9: Time slicing six 1-GPU jobs on 4 GPUs.

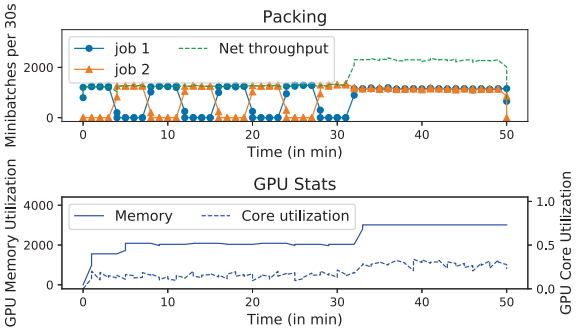


Figure 10: Packing jobs on single P40 GPU.

used for training are defaults from their references. All models take 6s or less per mini-batch in our evaluation. Thus, we set the time-slicing interval to 60s in these experiments.

6.1 Micro-benchmarks

In this section, we evaluate the *Gandiva* mechanisms, viz., time-slicing, packing, grow-shrink, and migration.

Time-slicing. We use six 1-GPU jobs on a single server with four P100 GPUs to illustrate time-slicing. These are ResNet-50 [24] models trained on the Cifar10 dataset using the PyTorch toolkit. When six 1-GPU jobs share four GPUs, each job ideally should get four minutes of GPU time out of every six.

Figure 9 shows a trace with the progress rate of each of the jobs over time. Initially, four 1-GPU long jobs are running and at time t=25min, two 1-GPU short jobs are scheduled at this server. One can see that the initial four 1-GPU jobs now get 4/6th their previous share. When the two short jobs depart, the long jobs return to their earlier performance. Also, note that the aggregate throughput of all jobs (right scale) is only marginally affected (less than 2%) during the entire trace, demonstrating that time-slicing is an efficient mechanism for providing early feedback during over-subscription.

Packing. Table 1 shows the performance of packing multiple jobs on a single GPU for various DLT mod-

Job	GPU Util (%)	Time Slicing (mb/s)	Packing Max (mb/s)	Packing Gain (%)
VAE [29]	8.7	81.8	419.3	412
SuperResolution [43]	14.1	40.3	145.2	260
RHN [58]	61.6	10.1	14.8	46
SCRNN [37]	66.8	16.7	23.3	39
MI-LSTM [52]	76.2	22.2	25.9	17
LSTM [5]	87.2	63.8	53.0	-16
ResNet-50 [24]	94.0	10.3	9.0	-13
ResNext-50 [53]	98.9	83.6	74.4	-11

Table 1: Packing multiple jobs on P40 (mb/s = minibatches/s).

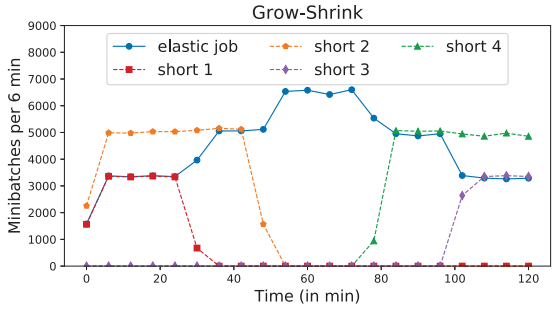


Figure 11: Grow from 1 to 4 GPUs, Shrink to 1-GPU.

els using PyTorch toolkit. For small DLT jobs with low GPU utilization, packing can provide significant gains of as much as 412%. For DLT jobs with middling GPU utilization, packing gains vary from model to model with some showing gains of up to 46%, but some exhibiting a loss of 16%. Finally, for image processing jobs with high utilization, such as ResNet-50 or ResNext-50 on the Cifar10 dataset, packing hurts performance by 11-13% .

Note that these packing results are without enabling NVIDIA's multi-process service (MPS) [7]. We found that MPS results in significant overhead in P40/P100 GPUs. However, hardware support for MPS in V100 GPUs [7] suggests that the use of MPS may be able to increase further packing gains in V100 GPUs.

Based on these results, predicting packing performance even with jobs of the same type appears challenging, let alone when jobs of different types are packed together. Instead, *Gandiva* adopts a profiling-based approach to packing. Figure 10 shows a case where two image super-resolution jobs [43] are initially being time-sliced on the same P40 GPU. After some time, the scheduler concludes that their memory and GPU core utilization is small enough that packing them is feasible and schedules them together on the GPU. The scheduler continues to profile their performance. Because their aggregate performance improves, packing is retained; otherwise (not shown), packing is undone and the jobs continue to use time-slicing.

Grow-Shrink. Grow-shrink is useful primarily when the cluster is under-utilized. *Gandiva* uses grow-shrink only for those jobs that specifically state that they can make use of this feature because users may want to adjust the learning rate and the batch size depending on the number of GPUs available. Figure 11 demonstrates this mechanism in action. Initially, a 4-P100 server has three jobs, 1-GPU growth-capable job, 1-GPU short job, and a 2-GPU short job, all using ResNet-50 with PyTorch. At time t=25min, the short job departs and after a time-out period of no new jobs being allocated to this GPU, the long running job expands to use 2 GPUs. At time t=45min, the second short job departs and the long running job expands to use all four GPUs. At time t=75min, a new 2-GPU job enters and the long job immediately shrinks to use two GPUs and, when another new 1-GPU job appears, the long job shrinks to use only 1 GPU. This micro-benchmark demonstrates that idle GPU resources can be effectively used with a mechanism like grow-shrink.

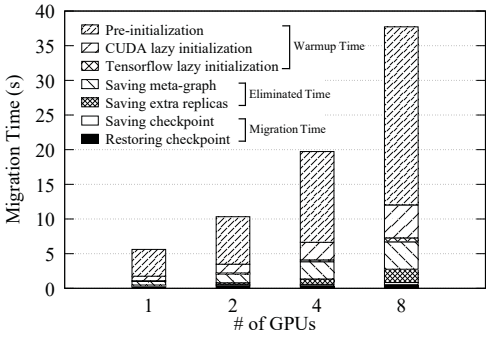


Figure 12: The breakdown of TF migration overhead.

Migration. We use a server with 8 P100 GPUs and the Tensorflow toolkit to evaluate migration overhead. First, we migrate a ResNet-50 single-server training job from one server to another. Figure 12 shows the detailed breakdown with a varying number of GPUs. Using our optimized implementation, we are able to eliminate or hide the majority of the migration overhead. The actual migration time, saving and restoring checkpoints, remains almost constant regardless of the number of GPUs because we save only one copy of the model. The loading of the in-memory checkpoint in each GPU runs in parallel and does not saturate the PCI-e bandwidth. The warm-up time and the cost due to meta-graph and checkpoints from other GPUs grow with the number of GPUs. As a result, we are able to save 98% of the migration overhead of 35s for 8-GPU jobs.

Figure 13 shows the max, min, and average intra-server and inter-server migration time of a 1-GPU job with 10 different deep learning models (summarized in Table 2) over 3 runs. Six of the 10 can be migrated within 1 second. Even the largest model (DeepSpeech [23] with a 1.4GB checkpoint) can be migrated in about 3.5 seconds, which is negligible compared to the long training time that often lasts for hours or days.

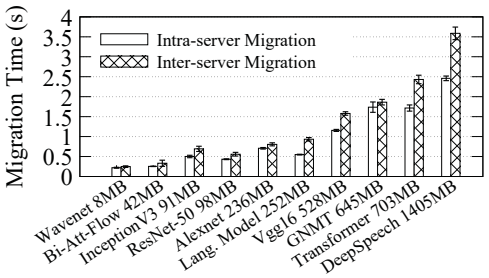


Figure 13: Migration time of real workloads.

6.2 Model exploration in a multi-job

AutoML, or automatic model exploration through hyper-parameter search is an important way to help users identify good neural models [21]. Typically, AutoML involves a hyper-parameter *configuration generator* and a *performance evaluator*. The generator uses different algorithms [11, 33] to generate new candidate configurations (DLT jobs), sometimes using the performance of prior configuration runs as a signal. The evaluator uses the early output of running jobs (*e.g.*, the learning curve) to predict the jobs' final performance and decide whether to continue running a given job or terminate it early.

Compared to a traditional scheduler where the number of configurations explored at any given time is limited by the number of GPUs available, *Gandiva* provides new primitives such as time-slicing and dynamic prioritization for AutoML algorithms to exploit. For example, the configuration generator is no longer limited by the number of GPUs and can dynamically generate many more configurations. Similarly, the performance evaluator can not only decide whether to continue or terminate a job but also how much priority to give to each configuration.

In this section, we explore one particular instance of using these new options enabled by *Gandiva* to highlight the potential benefit for AutoML. Detailed analysis of when and how many configurations to generate and/or how to best allocate priority among the various running configurations to utilize *Gandiva* features optimally is an open problem that we leave for future work.

At a high level, *Gandiva* can benefit an AutoML system in two ways. First, *Gandiva* can help AutoML explore more hyper-parameter configurations within a timespan, thereby enabling it to find better models [10, 19, 11]. Alternatively, *Gandiva* can help AutoML find a qualified model faster given a set of configurations through prioritization.

To demonstrate the benefit of *Gandiva* in exploring more configurations, we first use AutoML to run a multi-job to tune a LeNet-like CNN model with multiple convolution layers and fully connected layers, trained with the Cifar10 dataset. The hyper-parameters we search have 12 dimensions, including learning rate, dropout rate, number of layers, choice of optimization, etc. In this

	Neural model	Type	Dataset
10%	InceptionV3 [46]	CV	ImageNet [18]
	ResNet-50 [24]	CV	ImageNet
	Alexnet [31]	CV	ImageNet
	Vgg16 [44]	CV	ImageNet
60%	Bi-Att-Flow [42]	NLP	SQuAD [40]
	LanguageModel [56]	NLP	PTB [34]
	GNMT [51]	NLP	WMT16 [6]
	Transformer [49]	NLP	WMT16
30%	Wavenet [48]	Speech	VCTK [54]
	DeepSpeech [23]	Speech	CommonVoice [2]

Table 2: Neural models and the ratios in the trace.

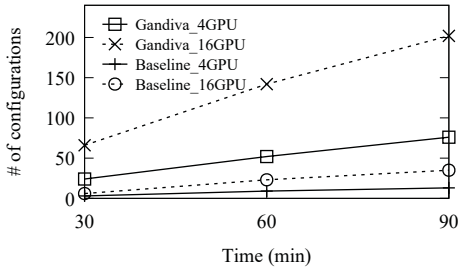


Figure 14: Model exploration number.

experiment, AutoML continually generates new hyper-parameter configurations based on Hyperopt [10] and leverages a curve-fitting method [19] to evaluate and predict the learning curve every 1,000 mini-batches (3% of the total mini-batches [19]). Jobs with no promise (less than 30% predicted accuracy) will be stopped early. The multi-job runs on 4 (or 16) P40 GPUs and each job requires 1 GPU. In the experiment, AutoML schedules 2 (or 8) more jobs every 1,000 mini-batches. In the baseline, jobs have to stay in a FIFO queue waiting for the running jobs to be terminated early or complete while in *Gandiva*, they are scheduled with time-slicing and migration support.

Figure 14 shows the number of explored hyper-parameter configurations. *Gandiva* can explore almost 10 times the number in the baseline approach in both the 4-GPU and 16-GPU cases. This is because, in the baseline approach, the GPUs can get "stuck" with a sub-optimal set of jobs that need to be run to completion, but in *Gandiva*, because of time-slicing, new configurations can be explored in parallel along with those jobs.

To demonstrate the benefit of *Gandiva* in finding a qualified model faster, we use Hyperopt to generate randomly the same set of 374 hyper-parameter configurations for both the baseline and *Gandiva*. The experiment measures the time required to find a configuration with at least 84% accuracy[2]. AutoML algorithms evaluate the jobs every 1,000 mini-batches and re-prioritize them based on the learning-curve prediction of their probability to achieve 84% accuracy [19]. In *Gandiva*, the top M jobs with the highest probabilities are then trained in the GPUs exclusively. In this experiment, we set M to 2 and 8 for 4-GPU and 16-GPU cases. Other jobs run in a time

[2]The LeNet-like CNN model is small: 84% is the best accuracy we found in the generated configurations.

	Position	93th (25%)	187th (50%)	280th (75%)	365th (98%)
4 GPUs	Baseline	691.5	1373.0	2067.2	2726.4
	Gandiva	125.5	213.8	302.4	387.1
	Speedup	5.51x	6.42x	6.84x	7.04x
16 GPUs	Baseline	253.0	492.7	731.7	970.0
	Gandiva	74.4	103.7	135.4	162.6
	Speedup	3.40x	4.75x	5.40x	5.96x

Table 3: Time to find a qualified configuration (minutes).

slicing manner. Our baseline approach stays the same as in the previous experiment. The result shows that *Gandiva* achieves 7x speedup compared to the baseline for the 4-GPU case and 6x for the 16-GPU case. More GPUs benefit the baseline as it implicitly improves the degree of parallelism of the long running jobs. There are two factors contributing to these gains. First, with prioritization, *Gandiva* grants more computation resources to the promising jobs. Second, because of the ability to run more configurations in parallel, *Gandiva* is able to find promising jobs quickly based on early feedback.

Further study shows the first job with the qualified configuration gets scheduled by *Gandiva* and the baseline in the 365^{th} place. We move the first qualified job from 365^{th} place to the first 25^{th} percentile, 50^{th} percentile, and 75^{th} percentile scheduling place and rerun the experiment. Table 3 summarizes the result: the later the qualified configuration shown, the larger gain *Gandiva* has. In a typical AutoML experiment, quality models usually show up later as those early-stopped jobs' configurations guide the system to find the better configurations.

To understand the sensitivity of *Gandiva*'s performance to the target accuracy of the model, we run AutoML with different target accuracies on a large state-of-the-art ResNet-like model (the official ResNet example in Keras [4]) for Cifar10. We use Hyperopt to generate 100 configurations, with the search space covering both the neural network architecture and various tunable hyper-parameters. The learning-curve prediction works as before; i.e., for every 3% of total mini-batches. The multi-job experiment runs on 16 P40 GPUs and every job runs on 1 GPU.

Table 4 shows the time spent on finding a model that is better than target accuracy using the baseline and *Gandiva* respectively. For a higher target accuracy, the performance gain of *Gandiva* is more notable. With 90% specified as a goal, the qualified model that is found achieves 92.62% validation accuracy. However, if the target accuracy is low; *e.g.*, 70%, a qualified model will appear early. In this case, the time for completing a single qualified configuration run dominates the total AutoML search time. Thus, *Gandiva* shows little benefit. We can see that when AutoML is used for achieving high accuracy models, *Gandiva* provides significant gains over the baseline.

Accuracy	70%	80%	90%
Baseline	134.1	2849.1	5296.7
Gandiva	134.1	543.1	935.4
Speedup	1.00x	5.25x	5.66x
Position	15th	58th	87th

Table 4: Model searching in ResNet-like network (minutes).

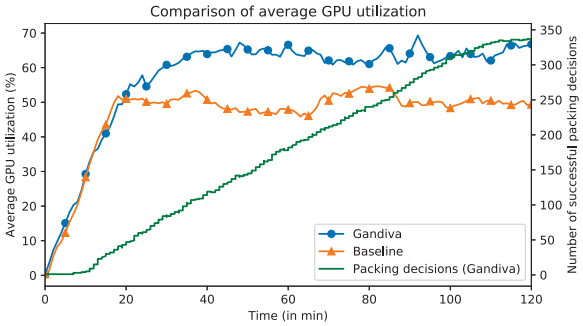


Figure 15: Cluster GPU utilization.

6.3 Cluster experiments: time-slicing and packing

In this section, we evaluate the *Gandiva* scheduler in a 45 server, 180-GPU cluster with about an equal mix of P100 and P40 GPUs. The scheduler implements both the reactive and introspective modes described earlier. In order to understand the gains contributed by different mechanisms in *Gandiva*, in this experiment, we only use time-slicing and packing, and disallow migrations. Further, none of the jobs are grow-shrink enabled. Thus, the accuracy achieved during training is unaffected by the *Gandiva* mechanisms.

We use the eight DLT jobs from Table 1 for this experiment and derive a mix of these jobs such that average GPU utilization is about 50%, similar to the average GPU utilization numbers reported from a study of a large deep learning cluster [28]. DLT jobs 1 and 2 from Table 1 (low utilization) are chosen with 0.3 probability, jobs 3, 4, and 5 (mid utilization) are chosen with 0.25 probability and jobs 6, 7, and 8 (high utilization) are chosen with probability of 0.45. Further, jobs 7 and 8 require either 2 or 4 GPUs while the rest each uses 1-GPU.

The number of mini-batches for each of these jobs are chosen such that, in isolation on P40, they take between 30 and 45 minutes of GPU time. A total of 1,000 jobs drawn from the above distribution arrive in a uniformly random manner over two hours. Using the same workload, we compare with *Gandiva* a baseline scheduler that does bin-packing but does not oversubscribe.

The primary goal of *Gandiva* is early feedback. We compute the average time to 100 mini-batches for all jobs as a measure of early feedback (*e.g.*, HyperBand [33] uses 100 mini-batches to evaluate a job). We find that the average time to complete 100 mini-batches is 498s for *Gandiva* and 2,203s for the baseline, for a reduction of 77%.

The second goal of *Gandiva* is cluster efficiency. Figure 15 shows the average GPU utilization of the cluster for the baseline scheduler and *Gandiva*, as well as the cumulative number of successful packings by *Gandiva* (right y-axis). The result shows clearly that *Gandiva* is able to use the cluster more efficiently than the baseline. The average utilization (computed over the stable regime from 20 to 200 mins) achieved by *Gandiva* is 62.8% compared to baseline average of 50.1%, resulting in a 26% relative improvement. Further, the greedy packing heuristic employed by *Gandiva* can be seen to be mostly successful with only a few packing decisions that need to be undone (the packing curve is mostly increasing with only small occasional dips).

6.4 Cluster experiments: time-slicing and migration

Trace. We collect a 9-day real-world job trace on a 2,000 GPU production cluster at Microsoft. The trace includes over 8,800 DLT jobs from three categories: computer vision (CV) (10%), Natural Language Processing (NLP) (60%), and Speech (30%), according to user survey and log analysis. However, the data/code used by these jobs are not available to us, due to security and privacy regulations. In their place, we pick 10 state-of-the-art deep learning models from Github with 50,000+ stars in total. The models are summarized in Table 2.

To synthesize a trace with similar characteristics as the production cluster, we mix these models with the same ratio as that in the trace. The number of mini-batches of the jobs in the trace are set to follow the job running time distribution of the 9-day real-world trace. We ensure that the synthesized trace closely follows the job running time distribution of the real-world trace, as shown in Figure 16. As before, none of the jobs are grow-shrink enabled in this experiment, as the cluster is in high load.

We run the trace using Hadoop's YARN capacity scheduler [50] and our *Gandiva* scheduler.

Fast-forwarding. To speed up replaying the 9-day trace, we leverage the predictability of the 10 models. We use the scheduler to instruct a running job to skip a number of mini-batches (i.e., fast-forwarding) whenever there are no scheduling events, including job arrival, departure, and migration, etc. The time skipped is calculated by measuring the previous mini-batch performance when the job reaches a stable state.

We validate fast-forwarding by constructing a 3-hour trace and compute average job completion time (JCT) and the makespan (the running time for the entire experiment) for the full trace and the experiment with fast-forwarding enabled using the capacity scheduler and *Gandiva*. The difference between the real and fast-forwarded experiment in all cases was less than 1%.

	Avg. JCT (mins)	Makespan (mins)
Cap. Sche.	832	13371
Gandiva	656	11349
Improvement	26.8%	17.8%

Table 5: Full trace experiment with fast-forwarding

Table 5 shows the average job completion time and the makespan for the two schedulers when replaying the synthesized job trace in a cluster with 100 GPUs (50 P100, 50 P40). We see that *Gandiva* improves average JCT by 26.8% and the total makespan is reduced by 17.8%. Figure 17 shows the CDF of the JCT of the two approaches: it shows *Gandiva* has more jobs with a JCT less than around 100 mins. During the entire experiment, *Gandiva* initiates migration 470 times; i.e., approximately once every 20 minutes.

Multi-job performance in a shared cluster. To compare the AutoML performance of a multi-job in a shared environment, we run the synthesized trace the same way as earlier in the same cluster with 100 GPUs. The trace-driven jobs act as background jobs, emulating a realistic shared cluster environment. At the $5,607^{th}$ minute (roughly in the middle of the trace), we launch two multi-jobs, each to find a qualified CNN model described in Section 6.2, trained on the Cifar10 dataset. Each multi-job is allocated 8 GPUs. For fair comparison, each multi-job is allowed to preempt other jobs to get 8 GPUs to reduce the unpredictable resource sharing.

We are particularly interested in understanding the effect of migration in *Gandiva* and therefore use a 2-GPU VGG-like model that is large and locality sensitive (Section 3.1). Each AutoML job runs for 100,000 mini-batches and reports the learning curve every 3,000 mini-batches (3%). Like the previous experiment, the job can be early stopped if the learning shows no promise [19]. In this experiment, the AutoML algorithm tunes the learning rate of the model with 40 configurations. The multi-job completes if a job's model achieves 99.5% training accuracy, with 91.3% validation accuracy. Again, the top M highest probability jobs run exclusively while other jobs are time-sliced. In this experiment, we set M to 2 (i.e., 4 GPUs).

As shown in Figure 18, with the capacity scheduler, it takes 1,215.74 and 1,110.62 mins, respectively, to find the qualified configuration for the two multi-jobs. *Gandiva*'s mechanisms like migration, time-slicing, and dynamic priority help provide better locality, identify promising jobs earlier, and improve the training speed of high priority jobs. As a result, *Gandiva* achieves a speedup of 13.6 and 12.9, respectively. Based on a micro-benchmark we did, we observed that time-slicing alone gave 7x gains for this AutoML experiment. Thus, the rest of the gains are attributable to improved locality due to migration. A real example of migration observed in this experiment was shown in Figure 8.

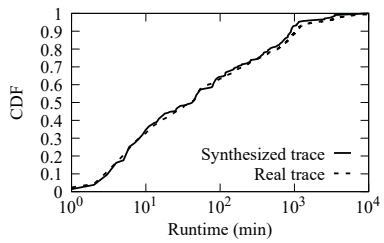


Figure 16: CDF of the synthesized trace

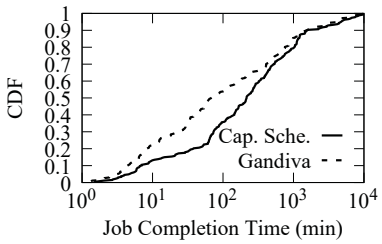


Figure 17: CDF of job completion time

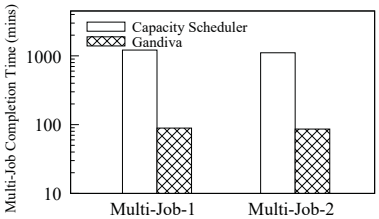


Figure 18: Multi-job completion time

7 Related Work

DLT job scheduling today. DLT jobs are scheduled today by big data schedulers such as Kubernetes or YARN [12, 28]. In these systems, a *fixed set of GPUs is assigned exclusively for the lifetime of a DLT job*. Thus, job queueing times can vary from a few minutes [12] to even hundreds of minutes [28] in these clusters.

An earlier study [28] shows that the average GPU utilization in a production cluster was only around 52%. Some jobs can inherently result in low GPU utilization due to the use of small models [29] and/or the use of small batch sizes for better generalization [35]. Further, jobs with inherently high GPU utilization can be adversely affected by poor GPU affinity and/or interference.

Scheduling policies for machine learning. Recent research [9, 15] also suggests that locality, interference, and GPU utilization are important performance factors for GPU workloads. They develop analytical models to predict the performance of GPU workloads. A *Gandiva* scheduler may leverage such models to guide its scheduling decisions. At its core, *Gandiva* framework is designed to empower DLT schedulers with the primitives such as time-slicing and migration.

SLAQ [57] proposes a scheduling policy that prioritizes resources in a CPU-based cluster to Spark jobs with high potentials (*e.g.*, the one with a fast improving learning curve). *Gandiva* can leverage the same policy for DLT on GPU clusters. Optimus [39] derives a proper number of parameter-servers and workers for MxNet-based deep learning jobs, which complements *Gandiva* in GPU cluster scheduling.

AutoML. *Gandiva* enables the co-design of DLT schedulers and AutoML algorithms like [10, 30]. Jobs in a multi-job can be promoted dynamically with more resource and/or better locality, accordingly to AutoML specific algorithms. Google Vizier [21], HyperDrive [41], and TuPAQ [45] focus more on the system design of AutoML. *Gandiva* empowers these systems with lower level system primitives that can further improve AutoML training experience in a multi-job, as shown in the experiments.

Big data cluster scheduling frameworks. Most recent big data scheduling frameworks assume jobs are modeled after a data flow graph (DFG) [26, 55, 13, 27, 20, 22]. Map/Reduce like tasks instantiated from the logical DFG get scheduled dynamically according to the job progress and the DFG dependency. *Gandiva* instead relies on the micro-task boundary implicitly defined by the mini-batch boundary. The low-level mechanisms of *Gandiva* such as time-slicing and migration also differ significantly from those big data scheduling systems [13, 22, 25, 14], while being surprisingly similar to a traditional operation system [47].

Time-slicing, suspend-resume, and process migration. *Gandiva* adopts traditional OS process primitives to facilitate DLT scheduling [47]. Unlike the general purpose OS mechanisms, *Gandiva* leverages the intra-job predictability of DLT to achieve a highly efficient implementation. *Gandiva* does not claim generality of the proposed techniques to other application domains.

8 Conclusion

We present *Gandiva*, a cluster scheduling framework for deep learning, which provides a set of efficient, low-level system primitives such as time-slicing, migration, intra-job elasticity, and dynamic priority. Using these primitives, *Gandiva* can effectively support neural model exploration in a multi-job, finding accurate neural models up to an order of magnitude faster than using traditional schedulers in a realistic shared cluster environment. *Gandiva* provides an efficient implementation of the proposed mechanisms by exploiting the intra-job predictability of DLT: our system prototype demonstrates that job suspend/resume and migration can be achieved under a second, even for cross-server migration for popular deep learning toolkits such as Tensorflow and PyTorch. Combined with an introspective scheduling policy, *Gandiva* improves overall cluster utilization by 26%.

Acknowledgments

We thank our shepherd KyoungSoo Park and the anonymous reviewers for their valuable comments and suggestions. We thank Bin Wang and Shuguang Liu from Bing search platform team and Daniel Li, Subir Sidhu, and Chandu Thekkath from Microsoft AI Platform team for providing access to the GPU clusters and Azure GPU VMs.

References

[1] Checkpoint/restore in user space. https://criu.org/Main_Page.

[2] Common voice dataset. https://voice.mozilla.org/.

[3] GlusterFS. https://docs.gluster.org/en/latest/.

[4] Keras. https://github.com/keras-team/keras.

[5] LSTM training on wikitext-2 dataset. https://github.com/pytorch/examples/tree/master/word_language_model.

[6] WMT16 dataset. http://www.statmt.org/wmt16/.

[7] Multi-process service, Oct 2017. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[8] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), vol. 16, USENIX Association, pp. 265–283.

[9] AMARAL, M., POLO, J., CARRERA, D., SEELAM, S., AND STEINDER, M. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 17.

[10] BERGSTRA, J., KOMER, B., ELIASMITH, C., YAMINS, D., AND COX, D. D. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery 8*, 1 (2015), 014008.

[11] BERGSTRA, J. S., BARDENET, R., BENGIO, Y., AND KÉGL, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554.

[12] BOAG, S., DUBE, P., HERTA, B., HUMMER, W., ISHAKIAN, V., JAYARAM, K., KALANTAR, M., MUTHUSAMY, V., NAGPURKAR, P., AND ROSENBERG, F. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS* (2017).

[13] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 285–300.

[14] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, omega, and kubernetes. *ACM Queue 14* (2016), 70–93.

[15] CHEN, Q., YANG, H., GUO, M., KANNAN, R. S., MARS, J., AND TANG, L. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 17–32.

[16] CHEN, T., XU, B., ZHANG, C., AND GUESTRIN, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[17] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113.

[18] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (2009), IEEE, pp. 248–255.

[19] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI* (2015), vol. 15, pp. 3460–8.

[20] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N. M., AND HAND, S. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 99–115.

[21] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), KDD '17, pp. 1487–1495.

[22] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 81–97.

[23] HANNUN, A., CASE, C., CASPER, J., CATANZARO, B., DIAMOS, G., ELSEN, E., PRENGER, R., SATHEESH, S., SENGUPTA, S., COATES, A., ET AL. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[24] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[25] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), vol. 11, pp. 22–22.

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[27] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 261–276.

[28] JEON, M., VENKATARAMAN, S., QIAN, J., PHANISHAYEE, A., XIAO, W., AND YANG, F. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *MSR-TR-2018-13* (2018).

[29] KINGMA, D. P., AND WELLING, M. Stochastic gradient VB and the variational auto-encoder. In *Second International Conference on Learning Representations, ICLR* (2014).

[30] KLEIN, A., FALKNER, S., SPRINGENBERG, J. T., AND HUTTER, F. Learning curve prediction with bayesian neural networks.

[31] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[32] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *nature 521*, 7553 (2015), 436.

[33] LI, L., JAMIESON, K., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).

[34] MARCUS, M. P., MARCINKIEWICZ, M. A., AND SANTORINI, B. Building a large annotated corpus of English: The penn treebank. *Computational linguistics 19*, 2 (1993), 313–330.

[35] MASTERS, D., AND LUSCHI, C. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612* (2018).

[36] MIIKKULAINEN, R., LIANG, J., MEYERSON, E., RAWAL, A., FINK, D., FRANCON, O., RAJU, B., SHAHRZAD, H., NAVRUZYAN, A., DUFFY, N., ET AL. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548* (2017).

[37] MIKOLOV, T., JOULIN, A., CHOPRA, S., MATHIEU, M., AND RANZATO, M. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753* (2014).

[38] PASZKE, A., GROSS, S., CHINTALA, S., AND CHANAN, G. PyTorch, 2017.

[39] PENG, Y., BAO, Y., CHEN, Y., WU, C., AND GUO, C. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems* (2018), ACM.

[40] RAJPURKAR, P., ZHANG, J., LOPYREV, K., AND LIANG, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[41] RASLEY, J., HE, Y., YAN, F., RUWASE, O., AND FONSECA, R. HyperDrive: Exploring hyperparameters with POP scheduling. In *Proceedings of the 18th International Middleware Conference, Middleware* (2017), vol. 17.

[42] SEO, M., KEMBHAVI, A., FARHADI, A., AND HAJISHIRZI, H. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603* (2016).

[43] SHI, W., CABALLERO, J., HUSZÁR, F., TOTZ, J., AITKEN, A. P., BISHOP, R., RUECKERT, D., AND WANG, Z. Real-time single image and video super-resolution using an efficient subpixel convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 1874–1883.

[44] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] SPARKS, E. R., TALWALKAR, A., HAAS, D., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 368–380.

[46] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2818–2826.

[47] TANENBAUM, A. *Distributed operating systems*. Prentice Hall, 1995.

[48] VAN DEN OORD, A., DIELEMAN, S., ZEN, H., SIMONYAN, K., VINYALS, O., GRAVES, A., KALCHBRENNER, N., SENIOR, A., AND KAVUKCUOGLU, K. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[49] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), pp. 6000–6010.

[50] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[51] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[52] WU, Y., ZHANG, S., ZHANG, Y., BENGIO, Y., AND SALAKHUTDINOV, R. R. On multiplicative integration with recurrent neural networks. In *Advances in Neural Information Processing Systems* (2016), pp. 2856–2864.

[53] XIE, S., GIRSHICK, R., DOLLÁR, P., TU, Z., AND HE, K. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (2017), IEEE, pp. 5987–5995.

[54] YAMAGISHI, J. English multi-speaker corpus for CSTR voice cloning toolkit, 2012. *URL http://homepages. inf. ed. ac. uk/jyamagis/page3/page58/page58. html*.

[55] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, ., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Eighth Symposium on Operating System Design and Implementation* (December 2008), USENIX.

[56] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[57] ZHANG, H., STAFMAN, L., OR, A., AND FREEDMAN, M. J. SLAQ: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SoCC '17, pp. 390–404.

[58] ZILLY, J. G., SRIVASTAVA, R. K., KOUTNIK, J., AND SCHMIDHUBER, J. Recurrent highway networks. In *International Conference on Machine Learning* (2017), pp. 4189–4198.

PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems

Yunseong Lee
Seoul National University

Alberto Scolari
Politecnico di Milano

Byung-Gon Chun
Seoul National University

Marco Domenico Santambrogio
Politecnico di Milano

Markus Weimer
Microsoft

Matteo Interlandi
Microsoft

Abstract

Machine Learning models are often composed of pipelines of transformations. While this design allows to efficiently execute single model components at training-time, prediction serving has different requirements such as low latency, high throughput and graceful performance degradation under heavy load. Current prediction serving systems consider models as black boxes, whereby prediction-time-specific optimizations are ignored in favor of ease of deployment. In this paper, we present PRETZEL, a prediction serving system introducing a novel white box architecture enabling both end-to-end and multi-model optimizations. Using production-like model pipelines, our experiments show that PRETZEL is able to introduce performance improvements over different dimensions; compared to state-of-the-art approaches PRETZEL is on average able to reduce 99th percentile latency by 5.5× while reducing memory footprint by 25×, and increasing throughput by 4.7×.

1 Introduction

Many Machine Learning (ML) frameworks such as Google TensorFlow [4], Facebook Caffe2 [6], Scikit-learn [48], or Microsoft ML.Net [14] allow data scientists to declaratively author pipelines of transformations to train models from large-scale input datasets. Model pipelines are internally represented as Directed Acyclic Graphs (DAGs) of operators comprising *data transformations* and *featurizers* (e.g., string tokenization, hashing, etc.), and *ML models* (e.g., decision trees, linear models, SVMs, etc.). Figure 1 shows an example pipeline for text analysis whereby input sentences are classified according to the expressed sentiment.

ML is usually conceptualized as a two-steps process: first, during *training* model parameters are estimated from large datasets by running computationally intensive iterative algorithms; successively, trained pipelines are used for *inference* to generate predictions through the estimated model parameters. When trained pipelines are served for inference, the full set of operators is deployed altogether. However, pipelines have different system characteristics based on the phase in which they are employed: for instance, at training time ML models run complex algorithms to scale over large datasets (e.g., linear models can use gradient descent in one of its many flavors [52, 50, 54]), while, once trained, they behave as other regular featurizers and data transformations; furthermore, during inference pipelines are often surfaced for direct users' servicing and therefore require low latency, high throughput, and graceful degradation of performance in case of load spikes.

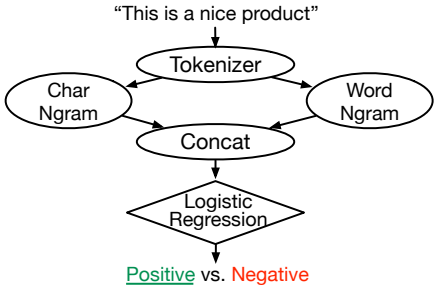


Figure 1: A Sentiment Analysis (SA) pipeline consisting of operators for featurization (ellipses), followed by a ML model (diamond). *Tokenizer* extracts tokens (e.g., words) from the input string. *Char* and *Word Ngrams* featurize input tokens by extracting n-grams. *Concat* generates a unique feature vector which is then scored by a *Logistic Regression* predictor. This is a simplification: the actual DAG contains about 12 operators.

Existing prediction serving systems, such as Clipper [9, 32], TensorFlow Serving [5, 46], Rafiki [59], ML.Net [14] itself, and others [17, 18, 43, 15] focus mainly on ease of deployment, where pipelines are con-

sidered as *black boxes* and deployed into *containers* (e.g., Docker [11] in Clipper and Rafiki, *servables* in TensorFlow Serving). Under this strategy, only "pipeline-agnostic" optimizations such as caching, batching and buffering are available. Nevertheless, we found that black box approaches fell short on several aspects. For instance, prediction services are profitable for ML-as-a-service providers only when pipelines are accessed in batch or frequently enough, and may be not when models are accessed sporadically (e.g., twice a day, a pattern we observed in practice) or not uniformly. Also, increasing model density in machines, thus increasing utilization, is not always possible for two reasons: first, higher model density increases the pressure on the memory system, which is sometimes dangerous—we observed (Section 5) machines swapping or blocking when too many models are loaded; as a second reason, co-location of models may increase tail latency especially when seldom used models are swapped to disk and later re-loaded to serve only a few users' requests. Interestingly enough, model pipelines often share similar structures and parameters inasmuch as A/B testing and customer personalization are often used in practice in large scale "intelligent" services; operators could therefore be shared between "similar" pipelines. Sharing among pipelines is further justified by how pipelines are authored in practice: ML pipelines are often produced by fine tuning pre-existing or default pipelines and by editing parameters or adding/removing steps like featurization, etc.

These and other limitations of existing black box systems (further described in Section 2) inspired us for developing PRETZEL: a system for serving predictions over trained pipelines originally authored in ML.Net and that borrows ideas from the Database and System communities. Starting from the above observation that trained pipelines often share operators and parameters (such as weights and dictionaries used within operators, and especially during featurization [64]), we propose a *white box* approach for model serving whereby end-to-end and multi-pipeline optimization techniques are applied to reduce resource utilization while improving performance. Specifically, in PRETZEL deployment and serving of model pipelines follow a two-phase process. During an *off-line phase*, statistics from training and state-of-the-art techniques from in-memory data-intensive systems [33, 66, 26, 40, 45] are used in concert to optimize and compile operators into *model plans*. Model plans are white box representations of input pipelines such that PRETZEL is able to store and re-use parameters and computation among similar plans. In the *on-line phase*, memory (data vectors) and CPU (thread-based execution units) resources are pooled among plans. When an inference request for a plan is received, an event-based scheduling [60] is used to bind computation to execution units.

Using 500 different production-like pipelines used internally at Microsoft, we show the impact of the above design choices with respect to ML.Net and end-to-end solutions such as Clipper. Specifically, PRETZEL is on average able to improve memory footprint by 25×, reduce the 99th percentile latency by 5.5×, and increase the throughput by 4.7×.

In summary, our contributions are:

- A thorough analysis of the problems and limitations burdening black box model serving approaches;
- A set of design principles for white box model serving allowing pipelines to be optimized for inference and to share resources;
- A system implementation of the above principles;
- An experimental evaluation showing order-of-magnitude improvements over several dimensions compared to previous black box approaches.

The remainder of the paper is organized as follows: Section 2 identifies a set of limitations affecting current black box model serving approaches; the outcome of the enumerated limitations is a set of design principles for white box model serving, described in Section 3. Section 4 introduces the PRETZEL system as an implementation of the above principles. Section 5 contains a set of experiments validating the PRETZEL performance, while Section 6 lists the limitations of current PRETZEL implementation and future work. The paper ends with related work and conclusions, respectively in Sections 7 and 8.

2 Model Serving: State-of-the-Art and Limitations

Nowadays, "intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML scoring capabilities, which are currently experiencing a growing demand [31]. This in turn fosters the research in prediction serving systems in cloud settings [5, 46, 9, 32], where trained models from data science experts are operationalized.

Data scientists prefer to use high-level declarative tools such as ML.Net, Keras [13] or Scikit-learn for better productivity and easy operationalization. These tools provide dozens of pre-defined operators and ML algorithms, which data scientists compose into sequences of operators (called *pipelines*) using high-level APIs (e.g., in Python).

ML.Net, the ML toolkit used in this paper, is a C# library that runs on a managed runtime with garbage collection and Just-In-Time (JIT) compilation. Unmanaged C/C++ code can also be employed to speed up processing when possible. Internally, ML.Net operators consume data vectors as input and produce one (or more) vectors as output. [1] Vectors are immutable whereby multiple downstream operators can safely consume the same input without triggering any re-execution. Upon pipeline initialization, operators composing the model DAG are analyzed and arranged to form a chain of function calls which, at execution time, are JIT-compiled to form a unique function executing the whole DAG on a single call. Although ML.Net supports Neural Network models, in this work we only focus on pipelines composed by featurizers and classical ML models (e.g., trees, logistic regression, etc.).

Pipelines are first trained using large datasets to estimate models' parameters. ML.Net models are exported as compressed files containing several directories, one per pipeline operator, where each directory stores operator parameters in either binary or plain text files. ML.Net, as other systems, aims to minimize the overhead of deploying trained pipelines in production by serving them into black box containers, where the same code is used for both training and inference. Figure 2 depicts a set of black box models where the invocation of the function chain (e.g., `predict()`) on a pipeline returns the result of the prediction: throughout this execution chain, inputs are pulled through each operator to produce intermediate results that are input to the following operators, similarly to the well-known Volcano-style iterator model of databases [36]. To optimize the performance, ML.Net (and systems such as Clipper among others) applies techniques such as handling multiple requests in batches and caching the results of the inference if some predictions are frequently issued for the same pipeline. However, these techniques assume no knowledge and no control over the pipeline, and are unaware of its internal structure. Despite being regarded as a good practice [65], the black box, container-based design hides the structure of each served model and prevents the system from controlling and optimizing the pipeline execution. Therefore, under this approach, there is no principled way neither for sharing optimizations between pipelines, nor to improve the end-to-end execution of individual pipelines. More concretely, we observed the following limitations in current state-of-the-art prediction serving systems.

Memory Waste: Containerization of pipelines disallows

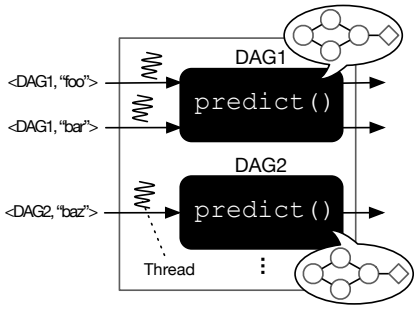


Figure 2: A representation of how existing systems handle prediction requests. Each pipeline is surfaced externally as a black box function. When a prediction request is issued (`predict()`), a thread is dispatched to execute the chain as a single function call.

any sharing of resources and runtimes [2] between pipelines, therefore only a few (tens of) models can be deployed per machine. Conversely, ML frameworks such as ML.Net have a known set of operators to start with, and featurizers or models trained over similar datasets have a high likelihood of sharing parameters. For example, transfer learning, A/B testing, and personalized models are common in practice; additionally, tools like ML.Net suggest default training configurations to users given a task and a dataset, which leads to many pipelines with similar structure and common objects and parameters. To better illustrate this scenario, we pick a Sentiment Analysis (SA) task with 250 different versions of the pipeline of Figure 1 trained by data scientists at Microsoft.

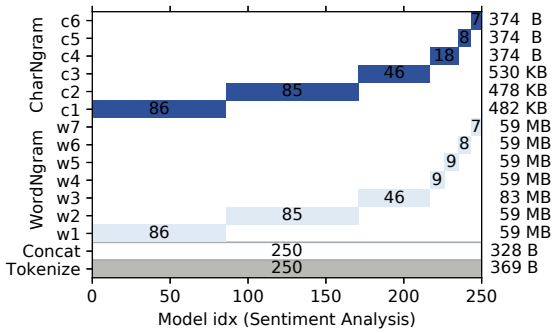


Figure 3: How many identical operators can be shared in multiple SA pipelines. CharNgram and WordNgram operators have variations that are trained on different hyper-parameters. On the right we report operators sizes.

Figure 3 shows how many different (parameterized) operators are used, and how often they are used within the 250 pipelines. While some operators like linear regression (whose weights fit in ~15MB) are unique to each pipeline,

[1] Note that this is a simplification. ML.Net in fact support several data types. We refer readers to [23] for more details.

[2] One instance of model pipeline in production easily occupies 100s of MB of main memory.

and thus not shown in Figure 3, many other operators can be shared among pipelines, therefore allowing more aggressive packing of models: Tokenize and Concat are used with the same parameters in all pipelines; Ngram operators have only a handful of versions, where most pipelines use the same version of the operators. This suggests that the resource utilization of current black box approaches can be largely improved.

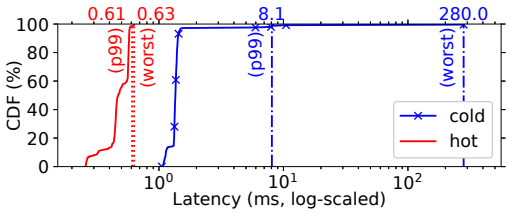


Figure 4: CDF of latency of prediction requests of 250 DAGs. We denote the first prediction as *cold*; the *hot* line is reported as average over 100 predictions after a warm-up period of 10 predictions. We present the 99th percentile and worst case latency values.

Prediction Initialization: ML.Net employs a pull-based execution model that lazily materializes input feature vectors, and tries to reuse existing vectors between intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection at training time. Conversely, this design forces memory allocation along the data path, thus making latency of predictions sub-optimal and hard to predict. Furthermore, at prediction time ML.Net deploys pipelines as in the training phase, which requires initialization of function chain call, reflection for type inference and JIT compilation. While this composability conveniently hides complexities and allows changing implementations during training, it is of little use during inference, when a model has a defined structure and its operators are fixed. In general, the above problems result in difficulties in providing strong tail latency guarantees by ML-as-a-service providers. Figure 4 describes this situation, where the performance of *hot* predictions over the 250 sentiment analysis pipelines with memory already allocated and JIT-compiled code is more than two orders of magnitude faster than the worst *cold* case version for the same pipelines.

To drill down more into the problem, we found that 57.4% of the total execution time for a single cold prediction is spent in pipeline analysis and initialization of the function chain, 36.5% in JIT compilation and the remaining is actual computation time.

Infrequent Accesses: In order to meet milliseconds-level latencies [61], model pipelines have to reside in main memory (possibly already warmed-up), since they can have MBs to GBs (compressed) size on disk, with loading and initialization times easily exceeding several seconds. A common practice in production settings is to unload a pipeline if not accessed after a certain period of time (e.g., a few hours). Once evicted, successive accesses will incur a model loading penalty and warming-up, therefore violating Service Level Agreement (SLA).

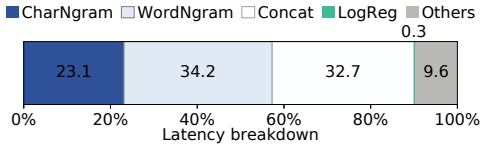


Figure 5: Latency breakdown of a sentiment analysis pipeline: each frame represents the relative wall clock time spent on an operator.

Operator-at-a-time Model: As previously described, predictions over ML.Net pipelines are computed by pulling records through a sequence of operators, each of them operating over the input vector(s) and producing one or more new vectors. While (as is common practice for in-memory data-intensive systems [45, 58, 24]) some interpretation overheads are eliminated via JIT compilation, operators in ML.Net (and in other tools) are "logical" entities (e.g., linear regression, tokenizer, one-hot encoder, etc.) with diverse performance characteristics. Figure 5 shows the latency breakdown of one execution of the SA pipeline of Figure 1, where the only ML operator (linear regression) takes two orders-of-magnitude less time with respect to the slowest operator (WordNgram). It is common practice for in-memory data-intensive systems to pipeline operators in order to minimize memory accesses for memory-intensive workloads, and to vectorize compute intensive operators in order to minimize the number of instructions per data item [33, 66]. ML.Net operator-at-a-time model [66] (as other libraries missing an optimization layer, such as Scikit-learn) is therefore sub-optimal in that computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline at hand, linear regression is commutative and associative (e.g., dot product between vectors) and can be pipelined with Char and WordNgram, eliminating the need for the Concat operation and the related buffers for intermediate results. As we will see in the following sections, PRETZEL's optimizer is able to detect this situation and generate an execution plan that is several times faster than the ML.Net version of the pipeline.

Coarse Grained Scheduling: Scheduling CPU resources carefully is essential to serve highly concurrent requests and run machines to maximum utilization. Under

the black box approach: (1) a thread pool is used to serve multiple concurrent requests to the same model pipeline; (2) for each request, one thread handles the execution of a full pipeline sequentially [3], where one operator is active at each point in time; (3) shared operators/parameters are instantiated and evaluated multiple times (one per container) independently; (4) thread allocation is managed by the OS; and (5) load balancing is achieved "externally" by replicating containers when performance degradation is observed. We found this design sub-optimal, especially in heavily skewed scenarios where a small amount of popular models are scored more frequently then others: indeed, in this setting the popular models will be replicated (linearly increasing the resources used) whereas containers of less popular pipelines will run underutilized, therefore decreasing the total resource utilization. The above problem is currently out-of-scope for black box, container-based prediction serving systems because they lack visibility into pipelines execution, and they do not allow models to properly share computational resources.

After highlighting the major inefficiencies of current black box prediction serving systems, we discuss a set of design principles for white box prediction serving.

3 White Box Prediction Serving: Design Principles

Based on the observations of Section 2, we argue that all previously mentioned limitations can be overcome by embracing a *white box approach* allowing to optimize the execution of predictions both horizontally *end-to-end* and vertically *among multiple model pipelines*.

White Box Prediction Serving: Model containerization disallows any sharing of optimizations, resources, and costs between pipelines. By choosing a white box architecture, pipelines can co-exist on the same runtime; unpopular pipelines can be maintained up and warm, while popular pipelines pay the bills. Thorough scheduling of pipelines' components can be managed within the runtime so that optimal allocation decisions can be made for running machines to high utilization. Nevertheless, if a pipeline requires exclusive access to computational or memory resources, a proper reservation-based allocation strategy can be enforced by the scheduler so that container-based execution can be emulated.

End-to-end Optimizations: The operationalization of models for prediction should focus on computation units making optimal decisions on how data are processed and results are computed, to keep low latency and gracefully degrade with load increase. Such computation units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator when possible, which are sensitive to branch mis-prediction and poor data locality [45]; and (3) avoid reflection and JIT compilation at prediction time. Optimal computation units can be compiled Ahead-Of-Time (AOT) since pipeline and operator characteristics are known upfront, and often statistics from training are available. The only decision to make at runtime is where to allocate computation units based on available resources and constraints.

Multi-model Optimizations: To take full advantage of the fact that pipelines often use similar operators and parameters (Figure 3), shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage. Similarly, execution units should be shared at runtime and resources properly pooled and managed, so that multiple prediction requests can be evaluated concurrently. Partial results, for example outputs of featurization steps, can be saved and re-used among multiple similar pipelines.

4 The Pretzel System

Following the above guidelines, we implemented PRETZEL, a novel white box system for cloud-based inference of model pipelines. PRETZEL views models as database queries and employs database techniques to optimize DAGs and improve end-to-end performance (Section 4.1.2). The problem of optimizing co-located pipelines is casted as a multi-query optimization and techniques such as view materialization (Section 4.3) are employed to speed up pipeline execution. Memory and CPU resources are shared in the form of vector and thread pools, such that overheads for instantiating memory and threads are paid upfront at initialization time.

PRETZEL is organized in several components. A *dataflow-style language integrated API* called Flour (Section 4.1.1) with related *compiler* and *optimizer* called Oven (Section 4.1.2) are used in concert to convert ML.Net pipelines into *model plans*. An Object Store (Section 4.1.3) saves and shares parameters among plans. A Runtime (Section 4.2.1) manages compiled plans and their execution, while a Scheduler (Section 4.2.2) manages the dynamic decisions on how to schedule plans based on machine workload. Finally, a FrontEnd is used to submit prediction requests to the system.

In PRETZEL, deployment and serving of model pipelines follow a two-phase process. During the *offline phase* (Section 4.1), ML.Net's pre-trained pipelines

[3] Certain pipelines allow multi-threaded execution, but here we evaluate only single-threaded ones to estimate the per-thread efficiency.

are translated into Flour transformations. Oven optimizer re-arranges and fuses transformations into model plans composed of parameterized logical units called *stages*. Each logical stage is then AOT-compiled into physical computation units where memory resources and threads are pooled at runtime. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared between pipelines with similar model plans. In the *on-line phase* (Section 4.2), when an inference request for a registered model plan is received, physical stages are parameterized dynamically with the proper values maintained in the Object Store. The Scheduler is in charge of binding physical stages to shared execution units.

Figures 6 and 7 pictorially summarize the above descriptions; note that only the on-line phase is executed at inference time, whereas the model plans are generated completely off-line. Next, we will describe each layer composing the PRETZEL prediction system.

4.1 Off-line Phase

4.1.1 Flour

The goal of Flour is to provide an intermediate representation between ML frameworks (currently only ML.Net) and PRETZEL, that is both easy to target and amenable to optimizations. Once a pipeline is ported into Flour, it can be optimized and compiled (Section 4.1.2) into a model plan before getting fed into PRETZEL Runtime for on-line scoring. Flour is a language-integrated API similar to KeystoneML [55], RDDs [63] or LINQ [42] where sequences of *transformations* are chained into DAGs and lazily compiled for execution.

Listing 1 shows how the sentiment analysis pipeline of Figure 1 can be expressed in Flour. Flour programs are composed by transformations where a one-to-many mapping exists between ML.Net operators and Flour transformations (i.e., one operator in ML.Net can be mapped to many transformations in Flour). Each Flour program starts from a `FlourContext` object wrapping the Object Store. Subsequent method calls define a DAG of transformations, which will end with a call to `Plan` to instantiate the model plan before feeding it into PRETZEL Runtime. For example, in lines 2 and 3 of Listing 1 the `CSV.FromText` call is used to specify that the target DAG accepts as input text in CSV format where fields are comma separated. Line 4 specifies the schema for the input data, where `TextReview` is a class whose parameters specify the schema fields names, types, and order. The successive call to `Select` in line 5 is used to pick the `Text` column among all the fields, while the call to `Tokenize` in line 6 is used to split the input fields into tokens. Lines 8 and 9 contain the two branches defining the char-level and word-level n-gram transformations, which are then merged with the `Concat` transform in lines 10/11 before the linear binary classifier of line 12. Both char and word n-gram transformations are parameterized by the number of n-grams and maps translating n-grams into numerical format (not shown in the Listing). Additionally, each Flour transformation accepts as input an optional set of statistics gathered from training. These statistics are used by the compiler to generate physical plans more efficiently tailored to the model characteristics. Example statistics are max vector size (to define the minimum size of vectors to fetch from the pool at prediction time, as in Section 4.2), dense/sparse representations, etc.

We have instrumented the ML.Net library to collect statistics from training and with the related bindings to the Object Store and Flour to automatically extract Flour programs from pipelines once trained.

Listing 1: Flour program for the SA pipeline. Parameters are extracted from the original ML.Net pipeline.

```
1  var fContext = new FlourContext(objectStore, ...)
2  var tTokenizer = fContext.CSV
3                   .FromText(',')
4                   .WithSchema<TextReview>()
5                   .Select("Text")
6                   .Tokenize();
7
8  var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);
9  var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);
10 var fPrgrm = tCNgram
11              .Concat(tWNgram)
12              .ClassifierBinaryLinear(cParams);
13
14 return fPrgrm.Plan();
```

4.1.2 Oven

With Oven, our goal is to bring query compilation and optimization techniques into ML.Net.

Optimizer: When `Plan` is called on a Flour transformation's reference (e.g., `fPrgrm` in line 14 of Listing 1), all transformations leading to it are wrapped and analyzed. Oven follows the typical rule-based database optimizer design where operator graphs (query plans) are transformed by a set of rules until a fix-point is reached (i.e., the graph does not change after the application of any rule). The goal of Oven Optimizer is to transform an input graph of Flour transformations into a stage graph, where each stage contains one or more transformations. To group transformations into stages we used the Tupleware's hybrid approach [33]: memory-intensive transformations (such as most featurizers) are pipelined together in a single pass over the data. This strategy achieves best data locality because records are likely to reside in

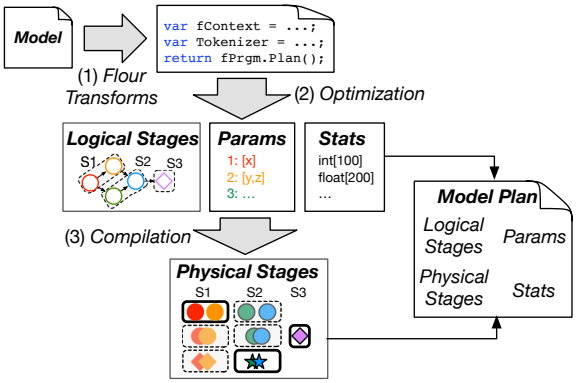


Figure 6: Model optimization and compilation in PRETZEL. In (1), a model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using logical stages, parameters, and statistics. A model plan is the union of all the elements.

CPU L1 caches [40, 45]. Compute-intensive transformations (e.g., vector or matrix multiplications) are executed one-at-a-time so that Single Instruction, Multiple Data (SIMD) vectorization can be exploited, therefore optimizing the number of instructions per record [66, 26]. Transformation classes are annotated (e.g., 1-to-1, 1-to-n, memory-bound, compute-bound, commutative and associative) to ease the optimization process: no dynamic compilation [33] is necessary since the set of operators is fixed and manual annotation is sufficient to generate properly optimized plans [4].

Stages are generated by traversing the Flour transformations graph repeatedly and applying rules when matching conditions are satisfied. Oven Optimizer consists of an extensible number of *rewriting steps*, each of which in turn is composed of a set of rules performing some modification on the input graph. Each rewriting step is executed sequentially: within each step, the optimizer iterates over its full set of rules until an iteration exists such that the graph is not modified after all rules are evaluated. When a rule is active, the graph is traversed (either top-down, or bottom up, based on rule internal behavior; Oven provides graph traversal utilities for both cases) and the rewriting logic is applied if the matching condition is satisfied over the current node. In its current implementation, the Oven Optimizer is composed of 4 rewriting steps:

InputGraphValidatorStep: This step comprises three rules, performing schema propagation, schema validation and graph validation. Specifically, the rules propagate schema information from the input to the final transformation in the graph, and validate that (1) each transformation's input schema matches with the transformation semantics (e.g., a WordNgram has a string type as input schema, or a linear learner has a vector of floats as input), and (2) the transformation graph is well-formed (e.g., a final predictor exists).

StageGraphBuilderStep: It contains two rules that rewrite the graph of (now schematized) Flour transformations into a stage graph. Starting with a valid transformation graph, the rules in this step traverse the graph until a pipeline-breaking transformation is found, i.e., a Concat or an n-to-1 transformation such as an aggregate used for normalization (e.g., L2). These transformations, in fact, require data to be fully scanned or materialized in memory before the next transformation can be executed. For example, operations following a Concat require the full feature vector to be available, or a Normalizer requires the L2 norm of the complete vector. The output of the `StageGraphBuilderStep` is therefore a stage graph, where each stage internally contains one or more transformations. Dependencies between stages are created as aggregation of the dependencies between the internal transformations. By leveraging the stage graph, PRETZEL is able to considerably decrease the number of vectors (and as a consequence the memory usage) with respect to the operator-at-a-time strategy of ML.Net.

StageGraphOptimizerStep: This step involves 9 rules that rewrite the graph in order to produce an optimal (logical) plan. The most important rules in this step rewrite the stage graph by (1) removing unnecessary branches (similar to common sub-expression elimination); (2) merging stages containing equal transformations (often generated by traversing graphs with branches); (3) inlining stages that contain only one transform; (4) pushing linear models through Concat operations; and (5) removal of unnecessary stages (e.g., when linear models are pushed through Concat operations, the latter stage can be removed if not containing any other additional transformation).

OutputGraphValidatorStep: This last step is composed of 6 rules. These rules are used to generate each stage's schema out of the schemas of the single internal transformations. Stage schema information will be used at runtime to request properly typed vectors. Additionally, some training statistics are applied at this step: transformations are labeled as sparse or dense, and dense compute-bound operations are labeled as vectorizable. A final validation check is run to ensure that the stage graph is well-formed.

In the example sentiment analysis pipeline of Figure

[4] Note that ML.Net does provide a second order operator accepting arbitrary code requiring dynamic compilation. However, this is not supported in our current version of PRETZEL.

1, Oven is able to recognize that the Linear Regression can be pushed into CharNgram and WordNgram, therefore bypassing the execution of Concat. Additionally, Tokenizer can be reused between CharNgram and WordNgram, therefore it will be pipelined with CharNgram (in one stage) and a dependency between CharNgram and WordNgram (in another stage) will be created. The final plan will therefore be composed of 2 stages, versus the initial 4 operators (and vectors) of ML.Net.

Model Plan Compiler: Model plans have two DAGs: a DAG of *logical stages*, and a DAG of *physical stages*. Logical stages are an abstraction of the results of the Oven Optimizer; physical stages contain the actual code that will be executed by the PRETZEL runtime. For each given DAG, there is a 1-to-n mapping between logical to physical stages so that a logical stage can represent the execution code of different physical implementations. A physical implementation is selected based on the parameters characterizing a logical stage and available statistics.

Plan compilation is a two step process. After the stage DAG is generated by the Oven Optimizer, the Model Plan Compiler (MPC) maps each stage into its logical representation containing all the parameters for the transformations composing the original stage generated by the optimizer. Parameters are saved for reuse in the Object Store (Section 4.1.3). Once the logical plan is generated, MPC traverses the DAG in topological order and maps each logical stage into a physical implementation. Physical implementations are AOT-compiled, parameterized, lock-free computation units. Each physical stage can be seen as a parametric function which will be dynamically fed at runtime with the proper data vectors and pipeline-specific parameters. This design allows PRETZEL runtime to share the same physical implementation between multiple pipelines and no memory allocation occurs on the prediction path (more details in Section 4.2.1). Logical plans maintain the mapping between the pipeline-specific parameters saved in the Object Store and the physical stages executing on the Runtime as well as statistics such as maximum vector size (which will be used at runtime to request the proper amount of memory from the pool). Figure 6 summarizes the process of generating model plans out of ML.Net pipelines.

4.1.3 Object Store

The motivation behind Object Store is based on the insights of Figure 3: since many DAGs have similar structures, sharing operators' state (parameters) can considerably improve memory footprint, and consequently the number of predictions served per machine. An example is language dictionaries used for input text featurization, which are often in common among many models and are relatively large. The Object Store is populated off-line by MPC: when a Flour program is submitted for planning, new parameters are kept in the Object Store, while parameters that already exist are ignored and the stage information is rewritten to reuse the previously loaded one. Parameters equality is computed by looking at the checksum of the serialized version of the objects.

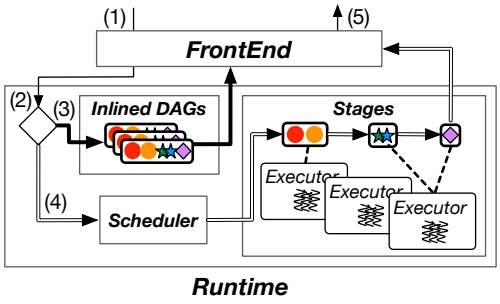


Figure 7: (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The FrontEnd returns the result to the Client once all stages are complete.

4.2 On-line Phase

4.2.1 Runtime

Initialization: Model plans generated by MPC are registered in the PRETZEL Runtime. Upon registration, a unique pipeline ID is generated, and physical stages composing a plan are loaded into a system *catalog*. If two plans use the same physical stage, this is loaded only once in the catalog so that similar plans may share the same physical stages during execution. When the Runtime starts, a set of vectors and long-running thread pools (called *Executors*) are initialized. Vector pools are allocated per Executor to improve locality [35]; Executors are instead managed by the Scheduler to execute physical stages (Section 4.2.2) or used to manage incoming prediction requests by the FrontEnd. Allocations of vector and thread pools are managed by configuration parameters, and allow PRETZEL to decrease the time spent in allocating memory and threads during prediction time.

Execution: Inference requests for the pipelines registered into the system can be submitted through the FrontEnd by specifying the pipeline ID, and a set of input records. Figure 7 depicts the process of on-line inference. PRET-

ZEL comes with a *request-response engine* and a *batch engine*. The request-response engine is used by single predictions for which latency is the major concern whereby context-switching and scheduling overheads can be costly. Conversely, the batch engine is used when a request contains a batch of records, or when the prediction time is such that scheduling overheads can be considered as negligible (e.g., few hundreds of microseconds). The request-response engine inlines the execution of the prediction within the thread handling the request: the pipeline physical plan is JIT-compiled into a unique function call and scored. Instead, by using the batch engine requests are forwarded to the Scheduler that decides where to allocate physical stages based on the current runtime and resource status. Currently, whether to use the request-response or batch engine is set through a configuration parameter passed when registering a plan. In the future we plan to adaptively switch between the two.

4.2.2 Scheduler

In PRETZEL, model plans share resources, thus scheduling plans appropriately is essential to ensure scalability and optimal machine utilization while guaranteeing the performance requirements.

The Scheduler coordinates the execution of multiple stages via a late-binding event-based scheduling mechanism similar to task scheduling in distributed systems [47, 63, 60]: each core runs an Executor instance whereby all Executors pull work from a shared pair of queues: one *low priority* queue for newly submitted plans, and one *high priority* queue for already started stages. At runtime, a scheduling event is generated for each stage with related set of input/output vectors, and routed over a queue (low priority if the stage is the head of a pipeline, high priority otherwise). Two queues with different priorities are necessary because of memory requirements. Vectors are in fact requested per pipeline (not per stage) and lazily fulfilled when a pipeline's first stage is being evaluated on an Executor. Vectors are then utilized and not re-added to the pool for the full execution of the pipeline. Two priority queues allow started pipelines to be scheduled earlier and therefore return memory quickly.

Reservation-based Scheduling: Upon model plan registration, PRETZEL offers the option to reserve memory or computation resources for exclusive use. Such resources reside on different, pipeline-specific pools, and are not shared among plans, therefore enabling container-like provision of resources. Note however that parameters and physical stage objects remain shared between pipelines even if reservation-based scheduling is requested.

4.3 Additional Optimizations

Sub-plan Materialization: Similarly to materialized views in database multi-query optimization [37, 29], results of installed physical stages can be reused between different model plans. When plans are loaded in the runtime, PRETZEL keeps track of physical stages and enables caching of results when a stage with the same parameters is shared by many model plans. Hashing of the input is used to decide whether a result is already available for that stage or not. We implemented a simple Least Recently Used (LRU) strategy on top of the Object Store to evict results when a given memory threshold is met.

External Optimizations: While the techniques described so far focus mostly on improvements that other prediction serving systems are not able to achieve due to their black box nature, PRETZEL FrontEnd also supports "external" optimizations such as the one provided in Clipper and Rafiki. Specifically, the FrontEnd currently implements prediction results caching (with LRU eviction policy) and delayed batching whereby inference requests are buffered for a user-specified amount of time and then submitted in batch to the Runtime. These external optimizations are orthogonal to PRETZEL's techniques, so both are applicable in a complementary manner.

5 Evaluation

PRETZEL implementation is a mix of C# and C++. In its current version, the system comprises 12.6K LOC (11.3K in C#, 1.3K in C++) and supports about two dozens of ML.Net operators, among which linear models (e.g., linear/logistic/Poisson regression), tree-based models, clustering models (e.g., K-Means), Principal Components Analysis (PCA), and several featurizers.

Scenarios: The goals of our experimental evaluation are to evaluate how the white box approach performs compared to black box. We will use the following scenarios to drive our evaluation:

- *memory*: in the first scenario, we want to show how much memory saving PRETZEL's white box approach is able to provide with respect to regular ML.Net and ML.Net boxed into Docker containers managed by Clipper.
- *latency*: this experiment mimics a request/response pattern (e.g., [19]) such as a personalized web-application requiring minimal latency. In this scenario, we run two different configurations: (1) a micro-benchmark measuring the time required by a system to render a prediction; and (2) an experiment measuring the total end-to-end latency observed by

Table 1: Characteristics of pipelines in experiments.

Type	Sentiment Analysis (SA)	Attendee Count (AC)
Input	Plain Text (variable length)	Structured Text (40 dimensions)
Size	50MB - 100MB (Mean: 70MB)	10KB - 20MB (Mean: 9MB)
Featurizers	N-gram with dictionaries (∼1M entries)	PCA, KMeans, Ensemble of multiple models

a client submitting a request.

- *throughput*: this scenario simulates a batch pattern (e.g., [8]) and we use it to assess the throughput of PRETZEL compared to ML.Net.
- *heavy-load*: we finally mix the above experiments and show PRETZEL's ability to maintain high throughput and graceful degradation of latency, as load increases. To be realistic, in this scenario we generate skewed load across different pipelines. As for the *latency* experiment, we report first the PRETZEL's performance using a micro-benchmark, and then we compare it against the containerized version of ML.Net in an end-to-end setting.

Configuration: All the experiments reported in the paper were carried out on a Windows 10 machine with 2 × 8-core Intel Xeon CPU E5-2620 v4 processors at 2.10GHz with Hyper Threading disabled, and 32GB of RAM. We used .Net Core version 2.0, ML.Net version 0.4, and Clipper version 0.2. For ML.Net, we use two black box configurations: a non-containerized one (1 ML.Net instance for all models), and a containerized one (1 ML.Net instance for each model) where ML.Net is deployed as Docker containers running on Windows Subsystem for Linux (WSL) and orchestrated by Clipper. We commonly label the former as just ML.Net; the latter as ML.Net + Clipper. For PRETZEL we AOT-compile stages using CrossGen [16]. For the end-to-end experiments comparing PRETZEL and ML.Net + Clipper, we use an ASP.Net FrontEnd for PRETZEL; the Redis front-end for Clipper. We run each experiment 3 times and report the median.

Pipelines: Table 1 describes the two types of model pipelines we use in the experiments: 250 unique versions of Sentiment Analysis (SA) pipeline, and 250 different pipelines implementing Attendee Count (AC): a regression task used internally to predict how many attendees will join an event. Pipelines within a category are similar: in particular, pipelines in the SA category benefit from sub-plan materialization, while those in the AC category are more diverse and do not benefit from it. These latter pipelines comprise several ML models forming an ensemble: in the most complex version, we have a dimensionality reduction step executed concurrently with a KMeans clustering, a TreeFeaturizer, and multi-class tree-based classifier, all fed into a final tree (or forest) rendering the prediction. SA pipelines are trained and scored over Amazon Review dataset [38]; AC ones are trained and scored over an internal record of events.

5.1 Memory

In this experiment, we load all models and report the total memory consumption (model + runtime) per model category. SA pipelines are large and therefore we expect memory consumption (and loading time) to improve considerably within this class, proving that PRETZEL's Object Store allows to avoid the cost of loading duplicate objects. Less gains are instead expected for the AC pipelines because of their small size.

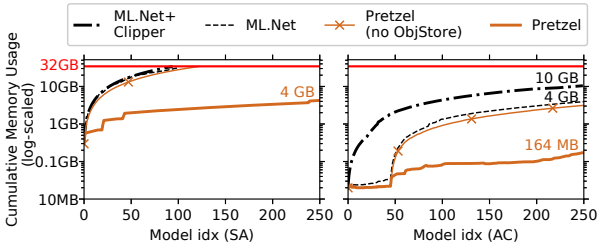


Figure 8: Cumulative memory usage (log-scaled) of the pipelines in PRETZEL, ML.Net and ML.Net + Clipper. The horizontal line represents the machine's physical memory (32GB). Only PRETZEL is able to load all SA pipelines within the memory limit. For AC, PRETZEL uses one order of magnitude less memory than ML.Net and ML.Net + Clipper. The memory usage of PRETZEL without Object Store is almost on par with ML.Net.

Figure 8 shows the memory usage for loading all the 250 model pipelines in memory, for both categories. For SA, only PRETZEL with Object Store enabled can load all pipelines. [5] For AC, all configurations are able to load the entire working set, however PRETZEL occupies only 164MBs: about 25× less memory than ML.Net and 62× less than ML.Net + Clipper. Given the nature of AC models (i.e., small in size), from Figure 8 we can additionally notice the overhead (around 2.5×) of using a container-based black box approach vs regular ML.Net.

[5] Note that for ML.Net, ML.Net + Clipper and PRETZEL without Object Store configurations we can load more models and go beyond the 32GB limit. However, models are swapped to disk and the whole system becomes unstable.

Keeping track of pipelines' parameters also helps reducing the time to load models: PRETZEL takes around 2.8 seconds to load 250 AC pipelines while ML.Net takes around 270 seconds. For SA pipelines, PRETZEL takes 37.3 seconds to load all 250 pipelines, while ML.Net fills up the entire memory (32GB) and begins to swap objects after loading 75 pipelines in around 9 minutes.

5.2 Latency

In this experiment we study the latency behavior of PRETZEL in two settings. First, we run a micro-benchmark directly measuring the latency of rendering a prediction in PRETZEL. Additionally, we show how PRETZEL's optimizations can improve the latency. Secondly, we report the end-to-end latency observed by a remote client submitting a request through HTTP.

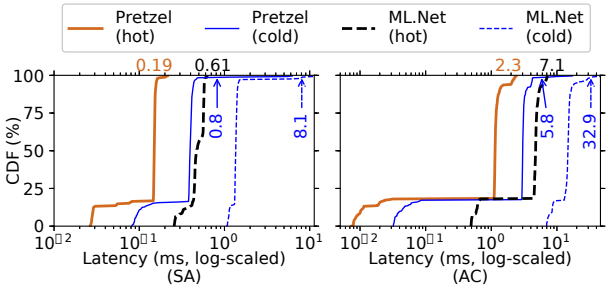


Figure 9: Latency comparison between ML.Net and PRETZEL. The accompanying blue lines represent the *cold* latency (first execution of the pipelines). On top are the P99 latency values: the hot case is above the horizontal line and the cold case is annotated with an arrow.

5.2.1 Micro-benchmark

Inference requests are submitted sequentially and in isolation for one model at a time. For PRETZEL we use the request-response engine over one single core. The comparison between PRETZEL and ML.Net for the SA and AC pipelines is reported in Figure 9. We start with studying *hot* and *cold* cases while comparing PRETZEL and ML.Net. Specifically, we label as cold the first prediction requested for a model; the successive 10 predictions are then discarded and we report hot numbers as the average of the following 100 predictions.

If we directly compare PRETZEL with ML.Net, PRETZEL is 3.2× and 3.1× faster than ML.Net in the 99th percentile latency in hot case (denoted by $P99_{hot}$), and about 9.8× and 5.7× in the $P99_{cold}$ case, for SA and AC pipelines, respectively. If instead we look at the difference between cold and hot cases relative to each system, PRETZEL again provides improvements over ML.Net. The $P99_{cold}$ is about 13.3× and 4.6× the $P99_{hot}$ in ML.Net, whereas in PRETZEL $P99_{cold}$ is around 4.2× and 2.5× from the $P99_{hot}$ case. Furthermore, PRETZEL is able to mitigate the long tail latency (worst case) of cold scoring. In SA pipelines, the worst case latency is 460.6× off the $P99_{hot}$ in ML.Net, whereas PRETZEL shows a 33.3× difference. Similarly, in AC pipelines the worst case is 21.2× $P99_{hot}$ for ML.Net, and 7.5× for PRETZEL.

To better understand the effect of PRETZEL's optimizations on latency, we turn on and off some optimizations and compare the performance.

AOT compilation: This options allows PRETZEL to preload all stage code into cache, removing the overhead of JIT compilation in the cold cases. Without AOT compilation, latencies of cold predictions increase on average by 1.6× and 4.2× for SA and AC pipelines, respectively.

Vector Pooling: By creating pools of pre-allocated vectors, PRETZEL can minimize the overhead of memory allocation at prediction time. When we do not pool vectors, latencies increase in average by 47.1% for hot and 24.7% for cold, respectively.

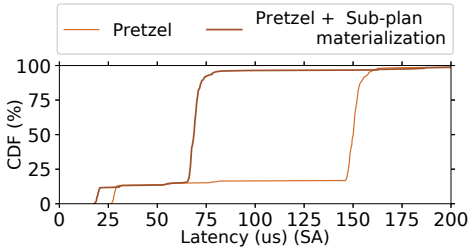


Figure 10: Latency of PRETZEL to run SA models with and without sub-plan materialization. Around 80% of SA pipelines show more than 2× speedup. Sub-plan materialization does not apply for AC pipelines.

Sub-plan Materialization: If different pipelines have common featurizers (e.g., SA as shown in Figure 3), we can further apply sub-plan materialization to reduce the latency. Figure 10 depicts the effect of sub-plan materialization over prediction latency for hot requests. In general, for the SA pipelines in which sub-plan materialization applies, we can see an average improvement of 2.0×, while no pipeline shows performance deterioration.

5.2.2 End-to-end

In this experiment we measure the end-to-end latency from a client submitting a prediction request. For PRETZEL, we use the ASP.Net FrontEnd, and we compare against ML.Net + Clipper. The end-to-end latency con-

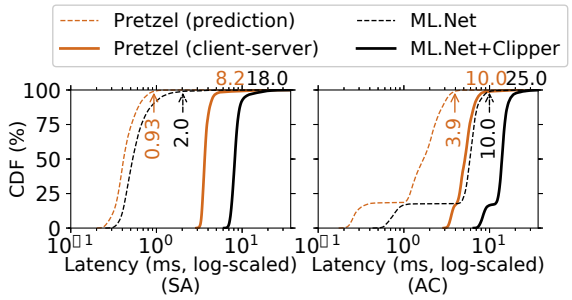


Figure 11: The latency comparison between ML.Net + Clipper and PRETZEL with ASP.Net FrontEnd. The overhead of client-server communication compared to the actual prediction is similar in both PRETZEL and ML.Net: the end-to-end latency compared to the just prediction latency is 9× slower in SA and 2.5× in AC, respectively.

siders both the prediction latency (i.e., Figure 9) as well as any additional overhead due to client-server communication. As shown in Figure 11, the latter overhead in both PRETZEL and ML.Net + Clipper is in the milliseconds range (around 4ms for the former, and 9 for the latter). Specifically, with PRETZEL, clients observe a latency of 4.3ms at *P*99 for SA models (vs. 0.56ms *P*99 latency of just rendering a prediction) and a latency of 7.3ms for AC models (vs. 3.5ms). In contrast, in ML.Net + Clipper, clients observe 9.3ms latency at *P*99 for SA models, and 18.0ms at *P*99 for AC models.

5.3 Throughput

In this experiment, we run a micro-benchmark assuming a batch scenario where all 500 models are scored several times. We use an API provided by both PRETZEL and ML.Net, where we can execute prediction queries in batches: in this experiment we fixed the batch size at 1000 queries. We allocate from 2 up to 13 CPU cores to serve requests, while 3 cores are reserved to generate them. The main goal is to measure the maximum number of requests PRETZEL and ML.Net can serve per second.

Figure 12 shows that PRETZEL's throughput (queries per second) is up to 2.6× higher than ML.Net for SA models, 10× for AC models. PRETZEL's throughput scales on par with the expected ideal scaling. Instead, ML.Net suffers from higher latency in rendering predictions and from lower scalability when the number of CPU cores increases. This is because each thread has its own internal copy of models whereby cache lines are not shared, thus increasing the pressure on the memory subsystem: indeed, even if the parameters are the same, the model objects are allocated to different memory areas.

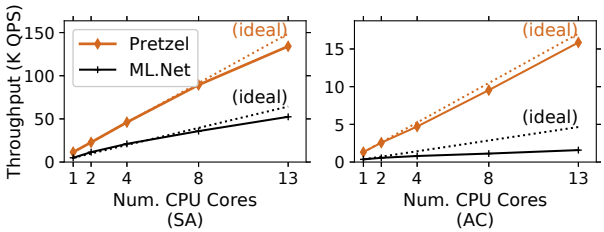


Figure 12: The average throughput computed among the 500 models to process one million inputs each. We scale the number of CPU cores on the x-axis and the number of prediction queries to be served per second on the y-axis. PRETZEL scales linearly to the number of CPU cores.

5.4 Heavy Load

In this experiment, we show how the performance changes as we change the load. To generate a realistic load, we submit requests to models by following the Zipf distribution ($\alpha = 2$).[6] As in Section 5.2, we first run a micro-benchmark, followed by an end-to-end comparison.

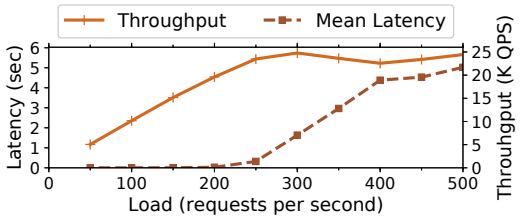


Figure 13: Throughput and latency of PRETZEL under the heavy load scenario. We maintain all 500 models in-memory within a PRETZEL instance, and we increase the load by submitting more requests per second. We report latency measurements from latency-sensitive pipelines, and the total system throughput.

5.4.1 Micro-benchmark

We load all 500 models in one PRETZEL instance. Among all models, we assume 50% to be "latency-sensitive" and therefore we set a batch size of 1. The remaining 50% models will be requested with 100 queries in a batch. As in the throughput experiment, we use the batch engine with 13 cores to serve requests and 3 cores to generate load. Figure 13 reports the average latency of latency-sensitive models and the total system throughput under different load configurations. As we increase the number of requests, PRETZEL's throughput increases linearly until it stabilizes at about 25k queries per second. Similarly, the average latency of latency-sensitive pipelines gracefully increases linearly with the load.

[6]The number of requests to the ith most popular models is proportional to $i^{-\alpha}$, where α is the parameter of the distribution.

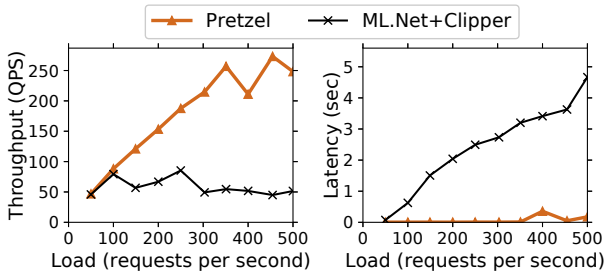


Figure 14: Throughput and latency of PRETZEL and ML.Net + Clipper under the end-to-end heavy load scenario. We use 250 AC pipelines to allow both systems to have all pipelines in memory.

Reservation Scheduling: If we want to guarantee that the performance of latency-critical pipelines is not degrading excessively even under high load, we can enable reservation scheduling. If we run the previous experiment reserving one core (and related vectors) for one model, this does not encounter any degradation in latency (max improvement of 3 orders of magnitude) as the load increases, while maintaining similar system throughput.

5.4.2 End-to-end

In this setup, we periodically send prediction requests to PRETZEL with the ASP.Net FrontEnd and ML.Net + Clipper. We assume all pipelines to be latency-sensitive, thus we set a batch of 1 for each request. As we can see in Figure 14, PRETZEL's throughput keeps increasing up to around 300 requests per second. If the load exceeds that point, the throughput and the latency begin to fluctuate. On the other hand, the throughput of ML.Net + Clipper is considerably lower than PRETZEL's and does not scale as the load increases. Also the latency of ML.Net + Clipper is several folds higher than with PRETZEL. The difference is due to the overhead of maintaining hundreds of Docker containers; too many context switches occur across/within containers.

6 Limitations and Future Work

Off-line Phase: PRETZEL has two limitations regarding Flour and Oven design. First, PRETZEL currently has several logical and physical stages classes, one per possible implementation, which make the system difficult to maintain in the long run. Additionally, different back-ends (e.g., PRETZEL currently supports operators implemented in C# and C++, and experimentally on FPGA [53]) require all specific operator implementations. We are however confident that this limitation will be overcome once code generation of stages will be added (e.g., with hardware-specific templates [41]). Secondly, Flour and Oven are currently limited to pipelines authored in ML.Net, and porting models from different frameworks to the white box approach may require non-trivial work. On the long run our goal is, however, to target unified formats such as ONNX [7]; this will allow us to apply the discussed techniques to models from other ML frameworks as well.

On-line Phase: PRETZEL's fine-grained, stage-based scheduling may introduce additional overheads in contrasts to coarse-grained whole pipeline scheduling due to additional buffering and context switching. However, such overheads are related to the system load and therefore controllable by the scheduler. Additionally, we found GC overheads to introduce spikes in latency. Although our implementation tries to minimize the number of objects created at runtime, in practice we found that long tail latencies are common. On white box architectures, failures happening during the execution of a model may jeopardize the whole system. We are currently working on isolating model failures over the target Executor. Finally, PRETZEL runtime currently runs on a single-node. An experimental scheduler adds Non Uniform Memory Access (NUMA) awareness to scheduling policies. We expect this scheduler to bring benefits for models served from large instances (e.g., [12]). We expect in the future to be able to scale the approach over distributed machines, with automatic scale in/out capabilities.

7 Related Work

Prediction Serving: As from the Introduction, current ML prediction systems [9, 32, 5, 46, 17, 30, 18, 43, 59, 15] aim to minimize the cost of deployment and maximize code re-use between training and inference phases [65]. Conversely, PRETZEL casts prediction serving as a database problem and applies end-to-end and multi-query optimizations to maximize performance and resource utilization. Clipper and Rafiki deploy pipelines as Docker containers connected through RPC to a front end. Both systems apply external model-agnostic techniques to achieve better latency, throughput, and accuracy. While we employed similar techniques in the FrontEnd, in PRETZEL we have not yet explored "best effort" techniques such as ensembles, straggler mitigation, and model selection. TensorFlow Serving deploys pipelines as *Servable*s, which are units of execution scheduling and version management. One Servable is executed as a black box, although users are allowed to split model pipelines and surface them into different Servables, similarly to PRETZEL's stage-based execution. Such optimization is however not automatic. LASER [22] enables large scale

training and inference of logistic regression models, applying specific system optimizations to the problem at hand (i.e., advertising where multiple ad campaigns are run on each user) such as caching of partial results and graceful degradation of accuracy. Finally, runtimes such as Core ML [10] and Windows ML [21] provide on-device inference engines and accelerators. To our knowledge, only single operator optimizations are enforced (e.g., using target mathematical libraries or hardware), while neither end-to-end nor multi-model optimizations are used. As PRETZEL, TVM [20, 28] provides a set of logical operators and related physical implementations, backed by an optimizer based on the Halide language [49]. TVM is specialized on neural network models and does not support featurizers nor "classical" models.

Optimization of ML Pipelines: There is a recent interest in the ML community in building languages and optimizations to improve the execution of ML workloads [20, 44, 27, 3, 39]. However, most of them exclusively target Neural Networks and heterogeneous hardware. Nevertheless, we are investigating the possibility to substitute Flour with a custom extension of Tensor Comprehension [57] to express featurization pipelines. This will enable the support for Neural Network featurizers such as word embeddings, as well as code generation capabilities (for heterogeneous devices). We are confident that the set of optimizations implemented in Oven generalizes over different intermediate representations.

Uber's Michelangelo [2] has a Scala DSL that can be compiled into bytecode which is then shipped with the whole model as a zip file for prediction. Similarly, H2O [1] compiles models into Java classes for serving. This is exactly how ML.Net currently works. Conversely, similar to database query optimizers, PRETZEL rewrites model pipelines both at the logical and at the physical level. KeystoneML [55] provides a high-level API for composing pipelines of operators similarly to Flour, and also features a query optimizer similar to Oven, albeit focused on distributed training. KeystoneML's cost-based optimizer selects the best physical implementation based on runtime statistics (gathered via sampling), while no logical level optimizations is provided. Instead, PRETZEL provides end-to-end optimizations by analyzing logical plans [33, 40, 45, 26], while logical-to-physical mappings are decided based on stage parameters and statistics from training. Similarly to the SOFA optimizer [51], we annotate transformations based on logical characteristics. MauveDB [34] uses regression and interpolation models as database views and optimizes them as such. MauveDB models are tightly integrated into the database, thus only a limited class of declaratively definable models is efficiently supported. As PRETZEL, KeystoneML and MauveDB provide sub-plan materialization.

Scheduling: Both Clipper [9] and Rafiki [59] schedule inference requests based on latency targets and provide adaptive algorithms to maximize throughput and accuracy while minimizing stragglers, for which they both use ensemble models. These techniques are external and orthogonal to the ones provided in PRETZEL. To our knowledge, no model serving system explored the problem of scheduling requests while sharing resource between models, a problem that PRETZEL addresses with techniques similar to distributed scheduling in cloud computing [47, 62]. Scheduling in white box prediction serving share similarities with operators scheduling in stream processing systems [25, 56] and web services [60].

8 Conclusion

Inspired by the growth of ML applications and ML-as-a-service platforms, this paper identified how existing systems fall short in key requirements for ML prediction-serving, disregarding the optimization of model execution in favor of ease of deployment. Conversely, this work casts the problem of serving inference as a database problem where end-to-end and multi-query optimization strategies are applied to ML pipelines. To decrease latency, we have developed an optimizer and compiler framework generating efficient model plans end-to-end. To decrease memory footprint and increase resource utilization and throughput, we allow pipelines to share parameters and physical operators, and defer the problem of inference execution to a scheduler that allows running multiple predictions concurrently on shared resources.

Experiments with production-like pipelines show the validity of our approach in achieving an optimized execution: PRETZEL delivers order-of-magnitude improvements on previous approaches and over different performance metrics.

Acknowledgments

We thank our shepherd Matei Zaharia and the anonymous reviewers for their insightful comments. Yunseong Lee and Byung-Gon Chun were partly supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2018-R0126-18-1093) supervised by the IITP (Institute for Information & communications Technology Promotion), and by the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test).

References

[1] H2O. https://www.h2o.ai/.

[2] Michelangelo. https://eng.uber.com/michelangelo/.

[3] TensorFlow XLA. https://www.tensorflow.org/performance/xla/.

[4] TensorFlow. https://www.tensorflow.org, 2016.

[5] TensorFlow serving. https://www.tensorflow.org/serving, 2016.

[6] Caffe2. https://caffe2.ai, 2017.

[7] Open Neural Network Exchange (ONNX). https://onnx.ai, 2017.

[8] Batch python API in Microsoft machine learning server, 2018.

[9] Clipper. http://clipper.ai/, 2018.

[10] Core ML. https://developer.apple.com/documentation/coreml, 2018.

[11] Docker. https://www.docker.com/, 2018.

[12] Ec2 large instances and numa. https://forums.aws.amazon.com/thread.jspa?threadID=144982, 2018.

[13] Keras. https://www.tensorflow.org/api_docs/python/tf/keras, 2018.

[14] ML.Net. https://dot.net/ml, 2018.

[15] MXNet Model Server (MMS). https://github.com/awslabs/mxnet-model-server, 2018.

[16] .Net Core Ahead of Time Compilation with CrossGen. https://github.com/dotnet/coreclr/blob/master/Documentation/building/crossgen.md, 2018.

[17] PredictionIO. https://predictionio.apache.org/, 2018.

[18] Redis-ML. https://github.com/RedisLabsModules/redis-ml, 2018.

[19] Request response python API in Microsoft machine learning server. https://docs.microsoft.com/en-us/machine-learning-server/operationalize/python/how-to-consume-web-services, 2018.

[20] TVM. https://tvm.ai/, 2018.

[21] Windows ml. https://docs.microsoft.com/en-us/windows/uwp/machine-learning/overview, 2018.

[22] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. LASER: A scalable response prediction platform for online advertising. In *WSDM*, 2014.

[23] Z. Ahmed and et al. Machine learning for applications, not containers (under submission), 2018.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *SIGMOD*, 2015.

[25] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[26] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. pages 225–237, 2005.

[27] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, 2015.

[28] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, 2018.

[29] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[30] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. In *CIDR*, 2015.

[31] D. Crankshaw and J. Gonzalez. Prediction-serving systems. *Queue*, 16(1):70:83–70:97, Feb. 2018.

[32] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[33] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, Aug. 2015.

[34] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[35] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, 1999.

[36] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[37] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[38] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, 2016.

[39] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing neural network queries over video at scale. *PVLDB*, 10(11):1586–1597, Aug. 2017.

[40] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[41] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[42] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[43] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque. TFX: A TensorFlow-based production-scale machine learning platform. In *SIGKDD*, 2017.

[44] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin. DyNet: The dynamic neural network toolkit. *ArXiv e-prints*, 2017.

[45] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, June 2011.

[46] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS*, 2017.

[47] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[49] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[50] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *NIPS*. 2011.

[51] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: an extensible logical optimizer for udf-heavy data flows. *Inf. Syst.*, 52:96–125, 2015.

[52] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, 2016.

[53] A. Scolari, Y. Lee, M. Weimer, and M. Interlandi. Towards accelerating generic machine learning prediction pipelines. In *IEEE ICCD*, 2017.

[54] S. Shalev-Shwartz and T. Zhang. Stochastic dual coordinate ascent methods for regularized loss. *J. Mach. Learn. Res.*, 14(1):567–599, Feb. 2013.

[55] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, 2017.

[56] T. Um, G. Lee, S. Lee, K. Kim, and B.-G. Chun. Scaling up IoT stream processing. In *APSys*, 2017.

[57] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, 2018.

[58] S. Wanderman-Milne and N. Li. Runtime code generation in cloudera impala. *IEEE Data Eng. Bull.*, 37:31–37, 2014.

[59] W. Wang, S. Wang, J. Gao, M. Zhang, G. Chen, T. K. Ng, and B. C. Ooi. Rafiki: Machine Learning as an Analytics Service System. *ArXiv e-prints*, Apr. 2018.

[60] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[61] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, 2015.

[62] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[63] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[64] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.*, 41(1):2:1–2:32, Feb. 2016.

[65] M. Zinkevich. Rules of machine learning: Best practices for ML engineering. https://developers.google.com/machine-learning/rules-of-ml.

[66] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage

Chinmay Kulkarni Sara Moore Mazhar Naqvi Tian Zhang Robert Ricci Ryan Stutsman
University of Utah

Abstract

In-memory key-value stores that use kernel-bypass networking serve millions of operations per second per machine with microseconds of latency. They are fast in part because they are simple, but their simple interfaces force applications to move data across the network. This is inefficient for operations that aggregate over large amounts of data, and it causes delays when traversing complex data structures. Ideally, applications could push small functions to storage to avoid round trips and data movement; however, pushing code to these fast systems is challenging. Any extra complexity for interpreting or isolating code cuts into their latency and throughput benefits.

We present *Splinter*, a low-latency key-value store that clients extend by pushing code to it. Splinter is designed for modern multi-tenant data centers; it allows mutually distrusting tenants to write their own fine-grained extensions and push them to the store at runtime. The core of Splinter's design relies on type- and memory-safe extension code to avoid conventional hardware isolation costs. This still allows for bare-metal execution, avoids data copying across trust boundaries, and makes granular storage functions that perform less than a microsecond of compute practical. Our measurements show that Splinter can process 3.5 million remote extension invocations per second with a median round-trip latency of less than 9 µs at densities of more than 1,000 tenants per server. We provide an implementation of Facebook's TAO as an 800 line extension that, when pushed to a Splinter server, improves performance by 400 Kop/s to perform 3.2 Mop/s over online graph data with 30 µs remote access times.

1 Introduction

Today's model of separated compute and storage is reaching its limits. Fast, kernel-bypass networking has yielded key-value stores that perform millions of requests per second per machine with microseconds of latency [22, 37, 45, 55, 71]. These systems gain much of their speed by being simple, allowing only lookups and updates. However, this simplicity results in inefficient data movement between storage and compute and costly client-side stalls [6, 51]. To efficiently exploit these new stores, applications will be under increasing pressure to push compute to them, but the granularity at which they can do so is a concern. At microsecond timescales, even small costs for isolation, containerization, or request dispatching dominate, placing practical limits on the granularity of functions that applications can offload to storage.

We resolve this tension in *Splinter*, a multi-tenant in-memory key-value store with a new approach to pushing compute to storage servers. Splinter preserves the low remote access latency (9 µs) and high throughput (3.5 Mops/s) of in-memory storage while adding native-code runtime *extensions* and the dense *multi-tenancy* (thousands of tenants) needed in modern data centers. Tenants send arbitrary type- and memory-safe extension code to stores at runtime, adding new operations, data types, or storage personalities. These extensions are exposed so tenants can remotely invoke them to perform operations on their data. Splinter's lightweight isolation lets thousands of untrusted tenants safely share storage and compute, giving them access to as much or as little storage as they need.

Splinter's design springs from the intersection of three trends: *in-memory storage with low-latency networking*, which is driving down the practical limits of request granularity; *massive multi-tenancy* driven by the cloud and the efficiency gains of consolidation; and *serverless computing*, which is already training developers to write stateless, decomposed application logic that can run anywhere in order to gain agility, scalability, and ease of provisioning.

Together, these trends drive Splinter's key design goals:

No-cost Isolation. Since extensions come from untrusted tenants, they must be isolated from one another. Hardware-based isolation is too expensive at microsecond time scales; even a simple page table switch would significantly impact response time and throughput.

Zero-copy Storage Interface. Extensions interact with stored data through a well-defined interface that serves as a trust boundary. For fine-grained requests, it must be lightweight in terms of transfer of control and in terms of data movement. This effectively requires extensions to be able to directly operate on tenant data *in situ* in the store, while maintaining protection and preventing data races with each other and the storage engine.

Lightweight Scheduling for Heterogeneous Tasks. Extensions are likely to be heterogeneous. Some extensions might involve simple point lookups of data or constructing small indexes; others might involve expensive computation or more data. Preemptive scheduling involves costly context switches, so Splinter must avoid preemption in the normal case, yet maintain it as an option to contain poorly-behaving extensions. It must also be able to support high quality of service under heavy skew, both in terms of the tenants issuing requests at different rates and extensions that take different amounts of time to complete.

Adaptive Multi-core Request Routing. With multiple tenants sharing a single machine, synchronization over tenant state can become a bottleneck. To minimize contention, tenants maintain locality by routing requests to preferred cores on Splinter servers. We can't, however, use a hard partitioning, as we don't want high skew to create hotspots and underused cores [58]. Routing decisions can't get in the way of fast dispatch of requests [7].

These goals give rise to Splinter's design. Developers write type-safe, memory-safe extensions in Rust [2] that they push to Splinter servers. Exploiting type-safety for lightweight isolation isn't new; SPIN [8] allowed applications to safely and dynamically load extensions into its kernel by relying on language-enforced isolation. Similarly, NetBricks [56] applied Rust's safety properties to dataplane packet processing to provide memory safety between sets of compile-time-known domains comprising network function chains. Splinter combines these approaches and applies them in a new and challenging domain. Language-enforced isolation with native performance and without garbage collection overheads is well-suited to low-latency data-intensive services like in-memory stores — particularly, when functionality must be added and removed at runtime by large numbers of fine-grained protection domains.

Splinter's approach allows it to scale to support thousands of tenants per machine, while processing more than 3.5 million tenant-provided extension invocations per second with a median response time of less than 9 µs. We describe our prototype of the Splinter key-value store and its extension and isolation model. We evaluate it on commodity hardware and show that a simple 800 line extension imbues Splinter with the functionality of Facebook's TAO [10]. On a single store, the extension can perform 3.2 million social graph operations per second with 30 µs average response times, making it competitive with the fastest known implementation [22].

2 Motivation

Splinter's key motivation is the desire to support complex data models and operations over large structures in a fast kernel-bypass stores. Existing in-memory stores trade data model for performance by providing a simple key-value interface that only supports `get` and `put`. Many real applications organize their data as trees, graphs, matrices, or vectors. Performing operations like aggregation or tree traversal with a key-value interface often requires multiple `gets`. Applications are usually *disaggregated* into a storage and compute tier, so these extra `gets` move data over the network and induce stalls for each request.

Figure 1 illustrates this problem with a storage client that traverses data logically organized as a tree. The client must first issue a `get` to retrieve the tree's root node. Next, it must perform a comparison and move down the tree by issuing another `get`. It must repeat this for every step of the traversal. Each `get` incurs a round trip that fetches a single node from storage; since the control flow is dependent on the data fetched, the client can only issue one request at a time. The number of round trips needed is proportional to the tree's depth, and a significant portion of the tree gets moved over the network. Even with modern low-latency networking, latency still dominates the client's performance: network transmission and processing takes tens of microseconds while the actual comparisons take less than a microsecond [55].

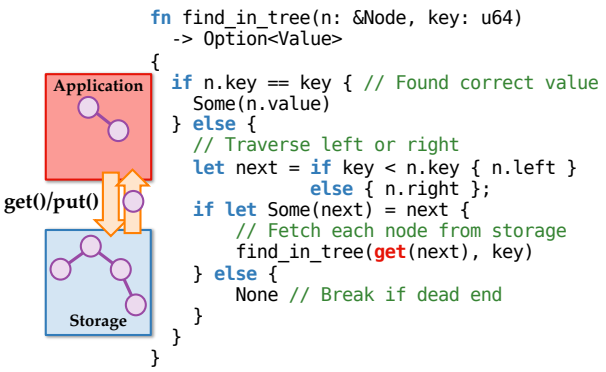


Figure 1: Tree traversal using `get()` operations over a key-value store. Each step requires a lookup at the storage layer, which is latency-bound and expensive for deep traversals. If multi-tenant stores could be safely extended this function could avoid remote access stalls and request processing costs.

One solution is to customize the storage tier of each application to support specialized data types. However, to improve efficiency and utilization, storage tiers are usually deployed as multi-tenant services [14, 19], so they cannot be customized for every possible data structure. SQL could be used at the storage tier, but SQL is known to be a poor fit for data types like graphs and matrices, does not support abstract data types, and is too expensive at microsecond timescales. Instead, Splinter takes a different approach; it allows applications to push small pieces of native compute (extensions) to stores at runtime. These extensions can implement richer data types and operators, avoiding extra round trips and reducing data movement.

2.1 The Need for Lightweight Isolation

Multi-tenancy at the storage layer makes running extensions challenging; a tenant cannot be allowed to access memory it does not own, starve others for resources, or crash the system. The major challenge is that, at microsecond timescales, context switches and data copying across isolation boundaries significantly hurt performance.

To quantify the overhead of hardware isolation, we simulated an 8-core multi-tenant store that isolates extensions using processes while varying the numbers of tenants making requests to it. Simulated requests con-

Xeon Architecture	Context switch delay (µs)	
	Pre KPTI	KPTI
D-1548, Broadwell	1.60	2.40
E5 2450, Sandy bridge	1.50	2.48
Gold 6142, Skylake	1.40	2.16

Table 1: Context switch overhead for different Intel Xeon architectures as measured on CloudLab. Each number represents the median of a million samples. Based on these measurements, we chose 2.16 µs and 1.40 µs for the context switch overhead with and without KPTI in our simulations.

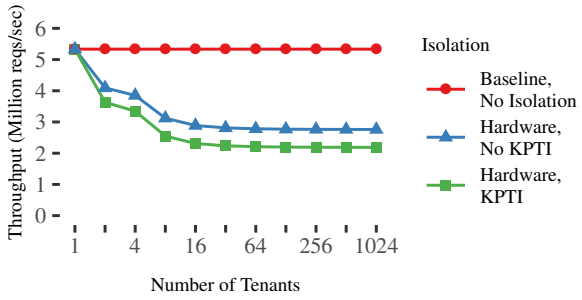


Figure 2: Simulated throughput versus the number of tenants. With hardware isolation, even modestly increasing the number of tenants to 16 (just twice the number of cores) leads to a significant drop in throughput. "No isolation" represents an upper bound where isolation costs are zero.

sume 1.5 µs of compute at the store; this is based on our benchmarks of simple unisolated operations on Splinter (§5.2); our numbers are similar to those reported by others' kernel-bypass stores [55]. Different context switch costs are simulated to show the overheads of hardware-based isolation of tenant code. The simulation only accounts for context switch costs; copying data across hardware isolation boundaries has also been shown to have significant performance costs [56]. Nearly all extensions will access data, which will force data copying when using hardware isolation and hurt throughput further. Based on measurements we made on different processor microarchitectures (Table 1), we simulate 1.40 µs of overhead for a basic context switch and 2.16 µs for a KPTI [16] protected kernel (which mitigates attacks that can leak the contents of protected memory [46]). The request pattern is uniform; all tenants make the same number of requests. The results are similar with skew. The simulator is also optimistic; whenever a request is made and an idle core is available at the store that last processed a request from the same tenant, the isolation cost is assumed to be zero.

Figure 2 presents simulated throughput at different tenant densities. The baseline represents an upper bound where extensions are run un-isolated at the storage system. The simulations show that throughput with hardware isolation (irrespective of KPTI) is significantly lower than the baseline. Even at just 16 tenants, context switch costs alone cut server throughput by a factor of 1.8.

Overall, for these types of fast stores, hardware isolation limits performance and tenant density. The challenges that we face in Splinter, and our design goals, stem from the need to (nearly) eliminate trust boundary crossing costs, to keep data movement across trust boundaries low, and to perform efficient fine-grained task scheduling.

3 Splinter Design

Each Splinter server works as an in-memory key-value store (Figure 3). Like most key-value stores, tenants can directly `get` and `put` values, but they can also customize the store at runtime by installing safe Rust-based extensions (shared libraries mapped into the store's address space) (Figure 3 ①). These extensions can define new operations on the tenant's data, including extensions that stitch together new data models in terms of the store's low-level get/put interface. Each tenant-provided extension is exported over the network, so a tenant can remotely invoke the procedures it has installed into the store.

Tenants send requests to a Splinter store over the network using kernel bypass (②). Splinter currently only supports a simple, custom UDP-based RPC protocol, though other optimized transports may provide similar performance [38]. Each tenant's requests are steered to a specific receive queue by the network card, improving locality (③). Each receive queue is paired with a single kernel thread (or *worker*) that is pinned to a specific core. Each worker pulls requests from its receive queue and creates a user-level task for the requested operation. Tasks provide an accounting context for resources consumed while executing the operation, the storage needed to suspend/resume the operation, and a unit of scheduling. Each worker has a task queue of new and suspended tasks, and it schedules across them to make progress in processing the operations (④). Scheduling is cooperative; as tasks yield and are resumed, they store/restore their state, so when a worker schedules a task no stack switch is performed. As tasks execute user-provided logic, they interact with the store through a `get`/`put` interface similar to the one exposed remotely (⑤); the key difference is that the functions exposed to extensions take and return references rather than forcing copies (Table 2).

Beyond fast kernel-bypass network request processing, Splinter's speed depends on exploiting the Rust compiler in two key ways: first, to enable low-cost isolation and, second, to enable low-cost task switching. The two are intertwined. Splinter uses stackless generators to suspend and resume running extensions, which require compiler support. That is, the Rust compiler analyzes extension code, determines the state that needs to be held across extension cooperative-yield/resume boundaries, and generates the code to suspend and resume extension operations. No separate stack is needed, and the code needed to yield/resume is transparent to the extension.

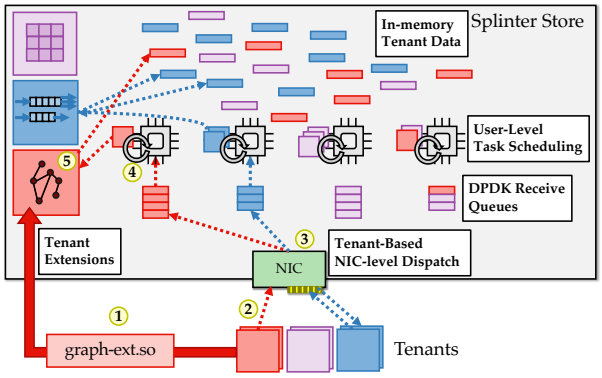


Figure 3: Overview of Splinter. Tenant data is stored in memory, and tenants can invoke extensions they have installed in the store (①). Extensions are type safe, but compile to native code. The NIC uses kernel bypass for low latency (②) and assists in dispatch by routing tenant requests to cores (③). Each core runs a single *worker* kernel thread that uses a user-level task scheduler to interleave the execution of tenant requests (④).

These lightweight tasks are key, but Splinter's careful attention to object lifetimes, ownership, and memory safety make them effective, since otherwise full context switch would be needed between tasks for isolation. A key challenge in Splinter is ensuring its fine-grained tasks from different trust domains—compiled to native code, and mapped directly into the store's memory—remain low-overhead while still operating within Rust's static safety checks. Low-overhead trust boundary crossings are essential to Splinter's design; they enable easy and inexpensive task switching, dispatch (§3.3), and work stealing (§3.4), which keep response latency low and CPU utilization high across all the cores of the store.

Another key challenge is that extension invocations introduce more irregularity into request processing than a simple get/put interface. By avoiding hardware context switches, Splinter keeps task switch costs down to about 11 nanoseconds, but the difficult tradeoff is that this forces it to handle these variable workloads without traditional preemptive scheduling. At the same time, it cannot use fully cooperative scheduling, since the store does not trust tenants to supply well-behaved extensions. Splinter's per-worker task scheduler resolves this tension by multiplexing long-running and short-running tasks to build mostly-cooperative scheduling. This is backed up by having an extra thread that acts as a watchdog for the others to support preemption when needed.

3.1 Compiling and Restricting Extensions

The Splinter store cannot directly load native code provided by tenants. Code must be compiled and type checked to ensure its safety before it can be loaded into a store, and extensions face some extra restrictions that must be enforced at compile time. The compiler is trusted and must be run by the storage provider. Tenants must not be able to tamper with the emitted extension, so it must be loaded directly into the store by the provider or the provider must ensure its integrity in transit between the trusted compiler and the store. Aside from Rust's standard type and lifetime checks (§3.1.2), Splinter extensions have the following static restrictions:

No Unsafe Code. Unsafe code could skip compiler checks resulting in memory unsafety. So, our wrapper over `rustc` disallows unsafe code in extensions (§3.1.3).

Module Whitelist. Code from external dependencies could include unsafe code, and that unsafe code shouldn't be incorporated into untrusted extensions unless it is trusted. Even beyond memory safety, such unsafe blocks could, for example, make syscalls. So, our wrapper restricts external dependencies to modules that are re-exported by a Splinter library that includes many standard functions and types. This restriction applies to the standard library (`std`) as well: the wrapper only exposes whitelisted `std` functionality to extensions.

These checks combine with three other runtime guarantees to ensure isolation: the store only accepts or provides references to insert/fetch a value under a key if the same tenant owns both the extension and the key (§3.2); it prevents uncooperative extensions from dominating CPU time and stack, heap, or record memory (§3.3); and it catches panics (runtime exceptions) and stack overflows that occur while executing an extension operation (§3.3). Next, we describe what guarantees this gives the storage provider and its tenants; the runtime checks are described later along with details about the execution model.

3.1.1 Trust Model

There are two stakeholders for a Splinter store: the storage provider and storage tenants. Splinter should protect tenants from each other and the provider from the tenants. Tenant misbehavior could be unintentional, in the form of bugs or unexpectedly high application load, or it could be malicious, in the form of tenants attempting to read others' data, deny service, or use an unfair fraction of resources. We consider threats from "within" the store; threats from "without" such as an attacker gaining root access to the machine by exploiting other services running on it should be dealt with using standard security best practices.

Aside from providing good quality of service to tenants, service providers have one key concern: protecting the secrecy and integrity of tenants' data. Extensions don't share state with one another, and Splinter provides no means for inter-extension communication. So, no complex sharing policies are needed; Splinter's only goal is extension isolation. Rust references act as capabilities; they ensure that extensions cannot fabricate arbitrary references to storage state or to other tenants' state (§3.1.2).

Like any database, Splinter's Trusted Computing Base (TCB) includes the libraries, compilers, hardware, etc. on

which it is built; while this code is not directly exposed to tenants, vulnerabilities in it can still lead to exploits. Dependencies include LLVM [42], the CPU, the network card (NIC) and its kernel-bypass libraries (DPDK [20]).

Splinter's design provides a larger attack surface relative to other databases in some ways, but decreases the attack surface in others. Because it allows execution of tenant code, Splinter's safety depends on the soundness of Rust's type system, which is not proven. While some soundness issues in the compiler have been found [34], progress is being made in proof efforts [35], and Splinter automatically benefits from such progress. If extensions cannot violate Rust's safe types, the remaining avenue for attack is unsafe code in the system; extensions cannot supply unsafe code, but they can indirectly call it in the interfaces and libraries that Splinter explicitly exposes to extensions. On the plus side, extensions *must* break one of these layers of protection before they can attack other code: they do not have direct access to system libraries, system calls, etc. and can only gain it by breaking out of Rust's safe environment.

Splinter decreases the attack surface with respect to the virtual memory system – both hardware and kernel components. Because it doesn't rely on virtual address translation for isolation, recent Meltdown speculation attacks don't affect its design [46]; however, Spectre-based speculation attacks do affect Splinter [40, 41]. Like any system that runs untrusted code or operates on untrusted inputs, Splinter would require special steps to mitigate these side channels. It already limits them in part because it doesn't provide explicit timing functions to extensions. Full protection will require compiler support [13], hardened storage interfaces (like the Linux kernel [17]), and hardened libraries for extensions. The measurements in this paper do not include these mitigations.

3.1.2 Memory Safety

Rust's memory safety (and data race freedom) is guaranteed through a strong notion of *ownership* that lets the `rustc` compiler reason statically about the lifetime of each object and any references to it. The compiler's *borrow checker* statically tracks where objects and references are created and destroyed. It ensures that the lifetime of a reference (initially determined by its binding's scope) is subsumed by the lifetime of its referent. Rust separates immutable and mutable references; an immutable reference is a reference that when held restricts access to the underlying object to be read-only. The compiler disallows multiple references (of either type) to an object while a mutable reference exists, which prevents data races.

Often, the lifetime of an object cannot be restricted to a single, static scope. This is especially true in a server that processes requests across threads, where the lifetime of many objects (RPC buffers, extension runtime state) is defined by request/response. Rust provides various accommodations for this, such as moving ownership between bindings and runtime reference counting that is safe but implemented in unsafe Rust. Splinter efficiently handles these issues while working within `rustc`'s static safety checks (§3.2.2). Unlike C/C++ pointers, Rust references cannot be fabricated or manipulated with arithmetic; they always refer to a valid, live object. Rust supports pointers but their use is restricted for safety.

Store Operations for Extensions

get(table: u64, key: &[u8]) → Option⟨ReadBuf⟩
Return view of current value stored under ⟨`table`, `key`⟩.

alloc(table: u64, key: &[u8], len: u64) → Option⟨WriteBuf⟩
Get buffer to be filled and then `put` under ⟨`table`, `key`⟩.

put(buf: WriteBuf) → bool
Insert filled buffer allocated with `alloc`.

args() → &[u8]
Return a slice to procedure args in request receive buffer.

resp(data: &[u8])
Append data to response packet buffer.

Table 2: Extensions interact with the store locally through an interface designed to avoid data copying.

3.1.3 Restricting Unsafe Rust

An important extra restriction that Splinter imposes beyond Rust is that extension code must be free from *unsafe* Rust, a superset of the language that allows operations that could violate its safety properties. For example, unsafe code can dereference pointers, perform unsafe casts, omit bounds checks, and implement low-level synchronization primitives. All unsafe code in Rust requires an `unsafe` block, which Splinter disallows in extension code.

Extensions cannot implement unsafe code, but they can invoke it indirectly. This is often desired. For example, extensions execute some unsafe code when they ask the store to populate a response packet buffer. In some cases it is not desired. For example, file I/O can be induced through the Rust standard library. To prevent this, Splinter restricts extensions to use a subset of the standard library that doesn't include I/O or OS functionality.

Our experience has been that safe Rust combined with basic data structures from its standard library are sufficient to write even complex imperative extensions like Facebook's TAO [10]. In cases where unsafe code could provide a performance benefit, the store can provide that functionality if it is deemed safe to do so, since it is trusted and can include `unsafe` code (§3.2.3).

3.2 Store Extension Interface

The interface that extensions use on the server to interact with stored records is similar to the external, remote interface that clients use in any conventional key-value store (Table 2). The main differences are in careful organization to eliminate the need to copy data between buffers.

All persisted records are stored in a *table heap*. Keeping records in a identifiable region will be essential to support replication, recovery, and garbage collection as Splinter's implementation evolves.

3.2.1 Storing Values

Extensions can `put()` data they receive over the network or new values that they produce into the store. When an extension invocation request is received from a tenant, the store invokes the indicated operation. Incoming data is in a packet buffer that is registered with the NIC. Those buffers cannot be used for long-term storage because the NIC must use them to receive new requests; data that must be preserved needs to be copied into the store.

Splinter tries to ensure that data can be moved from NIC buffers into the store with a single copy. This requires `put()` to be split into two steps. First, an extension calls `alloc(table, key, length)` to allocate a region in the table heap for a record. The extension receives a bounded slice (a view) to the underlying allocated memory. Then, it copies data from the request's receive buffer, unmarshalling as it does so, if needed. Extensions use `args()` to directly access data (by reference) in the receive buffer to perform this copy. An extension may produce its own data values as part of this process either from input arguments or together with values read from the store. Once the allocated region is properly populated, it is inserted into the table with `put()`, which takes ownership of the buffer and inserts it into a hash table.

Problems like use-after-free are prevented by Rust's borrow checker; extensions cannot hold references to a buffer once ownership is transferred to the store, eliminating the need for copying data into the store for safety. The receive packet buffer has the same guarantee. Rust's borrow checker ensures references to it cannot outlast the life of the RPC, eliminating the need to copy received arguments or data into the extension for safety.

Values stored by `put()` must be allocated from the table heap; extensions should not be able to pass arbitrary (heap or stack allocated) memory to `put()`. Splinter enforces this so that it can optimize record layout; keys and values can be forced into a single table heap allocation, which eases heap management and eliminates cache misses for hash table lookups. As a result, Splinter wraps allocations with a type (`WriteBuf`) that extensions cannot construct, ensuring they can only pass buffers acquired from `alloc()`. `WriteBuf` has a method to get a reference to the underlying buffer, so extensions can fill it.

```
fn aggregate(db: Rc<DB>) {
  let mut sum = 0u64;
  let mut status = SUCCESS;
  let key = &db.args()[..size_of::<u64>()];

  if let Some(key_lst) = db.get(TBL, key) {
    // Iterate KLEN sub-slices from key_lst
    for k in key_lst.read().chunks(KLEN) {
      if let Some(v) = db.get(TBL, k) {
        sum += v.read()[0] as u64;
      } else {
        status = INVALIDKEY;
        break;
      }
    }
  } else {
    status = INVALIDARG;
  }
  db.resp(pack(&status));
  db.resp(pack(&sum));
}
```

Listing 1: Example aggregate extension code. The extension takes a key as input (directly from a request receive buffer), looks it up in the store, and gets a reference to a value that contains a list of keys. It looks up each of those keys, it sums their values, and directly appends the result to a response buffer.

3.2.2 Accessing Values

Extensions can interact with stored data in a similar way, requiring only one copy into a response buffer to return values from the store. When an extension procedure is invoked, it is also provided with a response buffer that can be incrementally filled via `resp()`. On each extension procedure invocation, the store pre-populates the response buffer's packet headers; extensions can only append their data after these headers. All response buffers are pre-registered with the NIC for transmission.

Extensions call `get(table, key)`, and they receive back a reference to the underlying portion of the table heap that contains the value associated with `key`. No copying is needed at this step; the store tracks this reference and prevents the table heap garbage collector from freeing the buffer while an extension has a live reference to the data. Since values are never updated in place, extensions see stable views of values. Extensions can compute over the value or many values concurrently (by calling `get()` multiple times), and they can copy portions of the data they observe or any results they compute directly into the response buffer. Once the extension procedure has populated the response buffer, Intel's DDIO [32] transmits the data directly from the L1 cache, which avoids the cost of memory access for DMA of stored data.

Listing 1 and Figure 4 show an example of how this works for a simple extension that sums up a set of values stored under keys that are listed as part of another stored

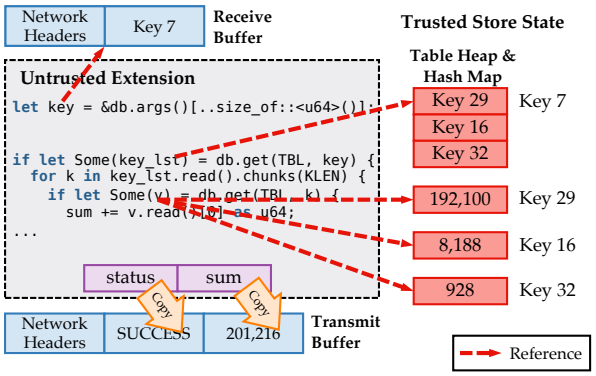


Figure 4: References during aggregation. All data accessed by the extension in Listing 1 is by reference whether that data is part of the arguments in the receive buffer or part of a record in the store. References work in reverse for the response; the extension passes references to data to the store, and the store copies that data into the response buffer.

value without any extra data copying. In Line 4, the extension obtains a reference to its transmit buffer to find which key it should look up in order to find a list of keys that will be aggregated over. Line 6 passes a reference to that same location to the store in order to obtain a reference to the value that contains the key list. In Line 8, still without copying, the extension iterates over that value in chunks equal to the length of the keys stored in the value. Each step of the iteration produces a reference that the extension uses to `get()` references to values for each of the stored keys, one at a time (Line 9). Using each of those references, it extracts a field that it adds to `sum`, a local variable. Finally, the extension passes references to `status` and `sum` to append them to the response buffer. In all, data copying is only forced where it is needed, so the compiler has flexibility in optimizing extension code.

The store's `get()` call returns a `ReadBuf` rather than a plain slice (`&[u8]`) in order to satisfy Rust's borrow checker. Calling `get()` cannot return an immutable reference or slice to a stored value, because the borrow checker wouldn't be able to statically verify that the reference would always refer to a valid location. For example, the compiler couldn't be sure that the store wouldn't garbage collect the value while the reference still exists. Furthermore, extension invocations are generators, and they must yield regularly (§3.3). Yielding marks the end and start of a new static scope, so each time the generator is resumed, the calling scope could vary. Any obtained references to a stored value couldn't be held across yields, because the borrow checker wouldn't be able to verify that those references would still be valid on reentry.

The `ReadBuf` returned by `get()` solves this. It is a smart pointer that maintains a reference count to ensure the underlying stored object isn't disposed, and it allows the extension code to (re-)obtain a reference to the underlying object data. Once a `ReadBuf` is returned to a generator, it is stored within the generator's local state, so the generator owns this `ReadBuf`. Extensions cannot hold references between yields, but by working with the `ReadBuf` it can (transparently) re-obtain a reference to the data without performing another `get()`. Rust's `Arc` smart pointer does the same; `ReadBuf` hides its constructor from extensions and disallows duplication. This prevents extension code from creating `ReadBuf`s that persist beyond the life of a single request/response, which could otherwise hold back table heap garbage collection.

3.2.3 Avoiding Serialization and De-serialization

Allowing extensions to interact directly with receive buffers, transmit buffers, and table heap buffers eliminates copying for opaque data, but Rust's safety makes avoiding some copies harder. Extensions cannot perform unsafe operations, otherwise they could thwart Rust's memory safety guarantees. Unfortunately, this means safe Rust code cannot cast an opaque byte array to/from different types to avoid the need to serialize/de-serialize data. For example, if `args()` returned an 8-byte slice an extension may desire to treat that slice data as a 64-bit unsigned value. Safe Rust disallows this.

For small arguments, extensions can convert between formats with arithmetic, but for richer data models, arguments, stored values, and responses will have more complex, structured formats. To accommodate this, Splinter's interface provides a mechanism for extension code to convert between byte slices and references to a small set of types. If a slice (`&[u8]`) is naturally aligned to the desired type, Splinter allows conversion to a reference of that type (`&T`), where `T` is limited to signed/unsigned integers and compound types built from them.

These casts are safe, but they are meaningless across architectures. As a result, they can only be used between a client and the store when they have the same underlying platform (e.g. x86-64). Similarly, they can only be used with extensions' `get`/`alloc`/`put` interface if all stores in the system (e.g. before/after recovery, source/destination for migration) have matching hardware platforms.

3.3 Cooperatively Scheduled Extensions

Splinter is designed to work well regardless of whether tenant-provided extensions are short and latency-sensitive or long-running and compute- or data-intensive. In fact, the best mix of tenants will mix these operations, keeping CPU, network, and in-memory storage better utilized than would be possible with a single, homogeneous workload. Even so, latency-sensitive operations can easily suffer under interference from heavier operations.

This means Splinter must multiplex execution of tenant extension invocations not only across cores but also within a core. Long-running procedures cannot be allowed to

dominate CPUs, but preemptive multitasking is too costly even when page table switching can be avoided.

Rust's lightweight isolation is part of the solution, since calls across trust domains have little overhead. Splinter already relies on `rustc` for safety, but it can also rely on it to help minimize task switching costs. When a new request comes into the store, Splinter calls into the responsible extension to allocate a stackless coroutine (a generator) that closes over the state needed to process the request. Generators support a `yield` statement that suspends execution and enables cooperative scheduling; extension code is expected to periodically call `yield` to allow other tasks to run. `rustc` produces generators specific to the extension, so the cost to create them and switch between them is low. Splinter invokes the created generator. Whenever it yields, Splinter's per-core task scheduler runs another generator task. Since yielding requires no costly hardware boundary crossing and no stack switch, it is fast and inexpensive to yield frequently.

Like other similar systems, to avoid jitter due to kernel thread context switches and migrations, Splinter runs the same number of worker threads as cores in the system (Figure 3), and each is pinned to a specific core. Generators are invoked on the worker's stack, avoiding a stack switch. Note that the compiler generates the structure to hold a suspended task's state across yields. Consequently, a worker's stack never concurrently contains state for different tenants (or even tasks); furthermore, whenever a task yields or completes, the worker's stack contains no extension state. This makes it easier to handle uncooperative extensions (§3.3.1) and load imbalance (§3.4).

3.3.1 Uncooperative and Misbehaving Extensions

All calls through the store interface include an implicit yield, so extensions can only dominate CPU time with infinite or compute-intensive loops. Nonetheless, such behavior can disrupt latency-sensitive tasks and constitute a denial-of-service attack in the limit.

To solve this, Splinter uses ideas from user-level threading for latency-sensitive services [59] and adapts them for untrusted code. An extra (mostly idle) thread acts as a watchdog. If a task on a core fails to yield for a few milliseconds, the watchdog remedies the situation. First, the worker thread on the core with the uncooperative task is re-pinned to a specific core that is shared among all misbehaving threads and low-priority background work that the store performs. Second, a new worker kernel thread is started and pinned to the idle core left behind after the misbehaving thread was re-pinned. Finally, the new worker steals the tasks remaining in the scheduler queue for the re-pinned worker and resumes execution for these tasks. Note, this is safe in part because all of the state of a suspended task is encapsulated. Tasks only have state on a worker's stack if they are running, so the misbehaving task is the only one the new worker cannot steal. Whenever a misbehaving task finally yields, the scheduler on that worker realizes that it has been displaced, and the worker thread terminates along with the task.

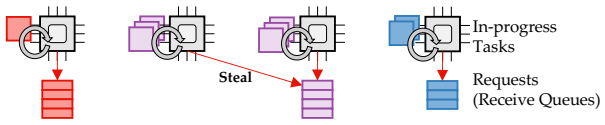


Figure 5: Dispatch tasks on each core steal requests from the receive queue of the core to their right whenever they have no requests in their own receive queue. As a result, work from overloaded cores get redistributed without generating high contention. Here, core 1's in-progress tasks were induced by requests stolen from core 2's queue.

Hence, misbehaving tasks don't block other requests, but they can still cause disruption. Creating and migrating kernel threads is expensive, so there must be a disincentive against forcing watchdog action. Tenants that run uncooperative tasks will experience poor quality of service, since they must share a core with other disruptive work. Furthermore, when a worker is re-pinned the watchdog also takes away access to its receive and transmit queues, so tenants cannot get responses from bad requests and, thus, benefit from their misbehavior. Even so, billing policies should ensure such behavior is unprofitable.

Aside from infinite loops, the store must also protect against other things that cannot be prevented with compile-time checks. For example, Rust doesn't have general exceptions, but extensions can raise exceptions with operations like division by zero that raise a `panic`. Splinter must "catch" these panics or they would terminate the worker, since panics unwind the call stack and worker threads call extension code on their own stack. Fortunately, Rust provides a mechanism to do this, and Splinter catches panics and converts them to an error response to the appropriate client. Stack overflows and violation of heap quotas are handled similarly.

3.4 Tenant Locality and Work Stealing

The Splinter store avoids any kind of centralized dispatch core to route requests to cores, since this can easily become a bottleneck [55]. At the same time, it needs to balance requests across cores, while still trying to exploit locality to avoid cross-core coordination overheads. To do this, clients route each tenant's requests to a particular core. This provides cache locality, it reduces contention, and it improves performance isolation. Splinter configures Flow Director [31] so that the NIC directly stores packets with a specific destination port number in a specific receive queue. Each receive queue is paired to a single task dispatcher owned by a worker thread (pinned to a core). As a result, tenants can steer requests to specific cores by placing their tenant id in the UDP destination port field.

CPU	2×Xeon E5-2640v4 2.40 GHz 10 cores (20 hardware threads) per socket
RAM	1 TB 2400 MHz DDR4
NIC	Mellanox CX5, 40 Gbps Ethernet
OS	Ubuntu 16.04, Linux 4.4.0-116, DPDK 17.08, 16×1 GB Hugepages, Rust 1.28.0-nightly

Table 3: Experimental configuration. Evaluation used one machine as server and one as client. Only the NIC-local CPU socket was used on the server.

However, this approach alone can leave cores idle under imbalance, and, as a multi-tenant store, it is important for the system to deliver good resource utilization. Whenever the scheduler on a core has no incoming requests in its local receive queue, it attempts to steal requests from a neighbor's receive queue (Figure 5). Transmit queues aren't bound to specific (server-side) source ports, so the response can be sent directly from the core that stole the request. This simple form of soft affinity works well, and, since tasks are lightweight, it is also relatively easy for Splinter to take advantage of idle compute in the system without costly thread migration.

4 Implementation

The Splinter store is implemented in 7,500 lines of Rust. It uses the NetBricks network function virtualization framework [56] as a wrapper over the DPDK [20] packet processing framework. Splinter also includes 1,100 lines of Rust that provide the store interface to extensions. Extensions import it and compile against it. The store also imports the interface, since it defines how the store interacts with extensions to create a new generator for an invocation. Splinter is open and freely available on github[1].

The store needn't be written in Rust, but doing so has advantages. It prevents data races and segmentation faults within the store, but it also lets the store use Rust's type system and lifetimes to ensure that mistakes aren't made with lifetimes of objects and references handed across trust boundaries, which an adversary could exploit.

5 Evaluation

We evaluated Splinter on five key questions:

1. What is Splinter's isolation overhead?
2. Does Splinter support high tenant densities?
3. How does Splinter perform under operations with heterogeneous runtimes?
4. Do representative extensions see latency and throughput benefits?
5. When does performing operations client-side outperform extension-based operations?

[1] https://github.com/utah-scs/Sandstorm/

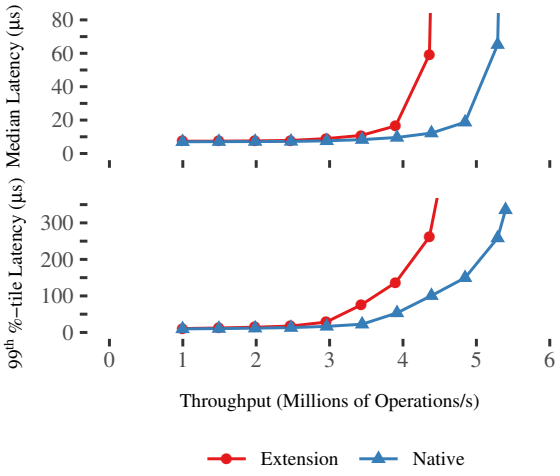


Figure 6: Comparison of YCSB-B performance using native and extension-based `get()` and `put()` operations at a tenant density of 1,024. When using extensions, the server saturates at 4.3 million operations per second. In comparison, native operations are about 23% more efficient, saturating at 5.3 million operations per second.

5.1 Experimental Setup

All evaluation was done on two machines consisting of one client and one storage server on the CloudLab testbed [60] (Table 3). Both used DPDK [20] over Ethernet using Mellanox NICs for kernel-bypass support. The server was configured to use only one processor socket; out of the ten hardware cores, eight were used for request processing, one was used for management and to detect misbehaving extensions, and the last one was used to hold all misbehaving extensions once detected.

To evaluate Splinter and its isolation costs under high load and density, the client ran a YCSB-B workload [15] (95% gets, 5% puts; keys were chosen from a Zipfian distribution with $\theta = 0.99$) that accessed tenant data on the storage server. Unless stated otherwise, the client simulates 1,024 total tenants. Tenant ids for each request were chosen from a Zipfian distribution with $\theta = 0.1$ (unless stated otherwise) to simulate some tenant skew. Each simulated tenant owns one data table consisting of 1 million 100 B record payloads with 30 B primary keys (totaling about 120 GB of stored data). The client always offered an open-loop load to the server.

5.2 Isolation Overhead

Figure 6 compares the performance of YCSB-B under two different cases. In one case ("Native"), the Splinter store executes get and put operations like any other key-value store would; none of Splinter's extension functionality is used. This case sets an upper-bound for Splinter's performance. In the other case ("Extension"), that same get or put is executed as part of a tenant-provided and untrusted

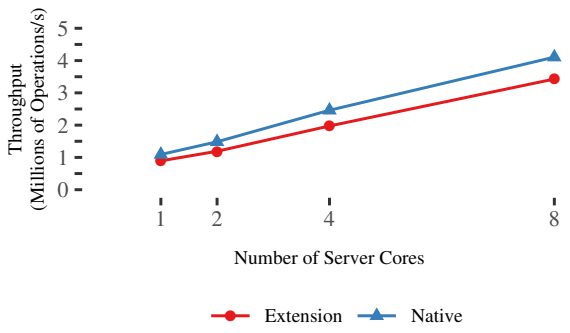


Figure 7: Storage server scalability at a tenant density of 1,024. Points represent throughput when YCSB-B latency crosses 10 µs. Isolation overhead is consistently lower than 20%.

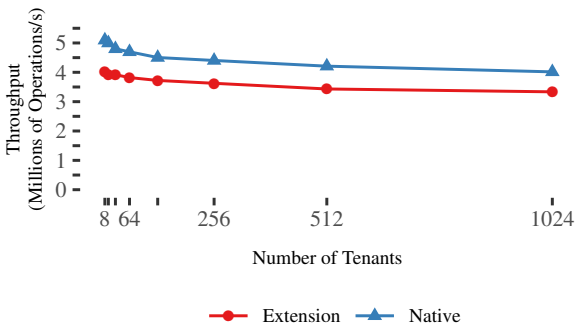


Figure 8: Scaling tenants. Points represent server throughput when YCSB-B latency crosses 10 µs. With isolation, increasing the number of tenants only impacts performance modestly; moving from 8 to 1,024 tenants reduces throughput by 700 Kops/s.

Splinter extension. This teases apart the isolation and dispatch costs for Splinter to run arbitrary tenant-provided logic. For offered loads of less than 3.5 million operations per second (Mops/s), median latency with and without isolation are nearly identical (about 9 µs).

Splinter extensions have some overhead, so the store saturates earlier when gets/puts are executed through extensions. With isolation, the median latency spikes above 4 Mops/s, reaching 59 µs at 4.3 Mops/s. Without isolation, this spike comes at 5.3 Mops/s. Tail latency (99^{th}-percentile) begins to show a difference at 3 Mops/s. On the whole, in this pessimal workload with extremely fine-grained operations all invoked as extensions, Splinter's isolation costs still only impact throughput of the store by about 19%. Compared to the 1.8× (simulated) penalty for hardware-based isolation in Figure 2, this is a significant improvement (a 1.2× penalty over native get/put).

Figure 7 compares YCSB-B scalability when the server is approaching saturation (median latency $>$ 10 µs) under the native and extension-based cases. Invoking get and put operations from extensions instead of directly has no impact on scalability; scalability is near linear in both scenarios. However, as pointed out above, it does affect throughput. At one core, throughput is reduced by 200 Kops/s (18%), while at eight cores, the reduction is 700 Kops/s (17%). This shows that, though extensions do increase the number of cycles each core spends processing requests, it doesn't come at the cost of significant increased coordination between the cores.

5.3 Tenant Density

Figure 8 shows how varying the number of tenants sharing the store impacts its throughput. As in the prior experiments, tenants run YCSB-B under two cases: without isolation ("Native") and with isolation ("Extension"), so the experiment captures extension isolation overheads. The results show that Splinter can efficiently support high tenant densities with minimal overhead. With isolation, the throughput at 1,024 tenants is 3.3 Mops/s, only 700 Kops/s less than the throughput at 8 tenants. Additionally, the throughput with isolation is consistently within 22% of the throughput without isolation.

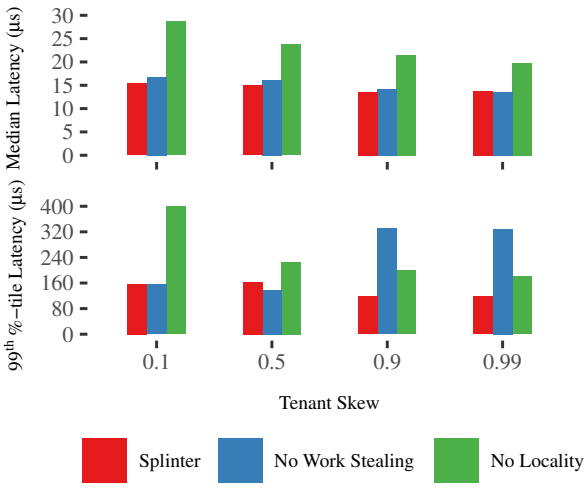


Figure 9: Latency with tenant skew. The server runs near saturation at 4 Mops/s in each case. Without work stealing, tail latency under high skew increases from 138 µs to 330 µs. Without tenant locality, median and tail latencies are affected.

In practice, offered tenant load will be skewed, since some tenants are likely to have heavier workloads than others. This results in a few heavy workloads that must share the store with a long tail of many more passive ones. We ran an experiment to show that Splinter can handle this imbalance and that its work stealing and tenant locality help maintain Splinter's response times under high load.

Recall that Splinter routes requests for a tenant to a specific core, but cores steal work from each other to combat imbalance. To gauge the benefits of this approach, we compare it against a tenant-partitioned approach with no work stealing and an unpartitioned approach that sprays requests over all cores in a tenant-oblivious fashion. We vary tenant skew, which affects all three approaches.

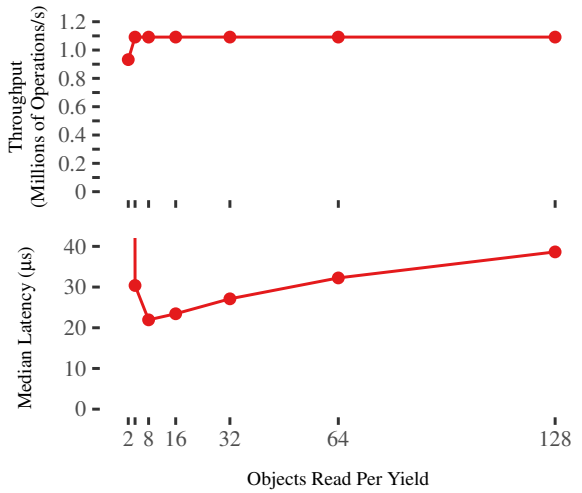


Figure 10: Performance with a small fraction (15%) of cooperative long running procedures that perform 128 `gets`. Yielding frequently can help improve median latency from 38 µs to 22 µs. However, yielding too frequently hurts median latency. The storage server was offered a constant load of 1.1 Mops/s.

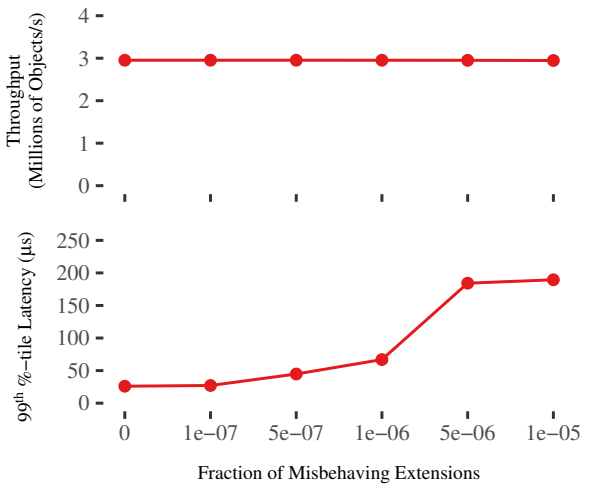


Figure 11: Impact of uncooperative requests on performance. System throughput stays constant at 3 Mops/s throughout. For fractions of uncooperative requests greater than 1 every million, tail latency is significantly affected (> 100 µs).

Figure 9 shows the results. These measurements are with an offered load of 4 Mops/s, keeping the store close to saturation. In each case, the store meets the offered load by running at 4 Mop/s. Without work stealing, Splinter's tail latency suffers by a factor of 2 under high tenant skew (0.9 and 0.99). In this case, partitioning helps throughput due to locality and reduced contention (as evidenced by its relatively consistent median response time), but queues become imbalanced hurting tail latency. The unpartitioned approach doesn't respond as significantly to tenant skew though it is slower overall, as expected. Unpartitioned execution results in 42% to 86% worse median latency with 38% to 155% worse tail latency.

5.4 Request Heterogeneity

Figure 10 investigates the impact of mixing short operations with cooperative longer-running operations. We configured our client so that 15% of extension operations performed 128 `gets` on the storage server. The rest of the requests invoked an extension that performed one `get`. We varied the number of `gets` made by the longer extension per yield (frequency). These measurements were made at an offered load of 1.1 Mops/s. Increasing the frequency of yields improves median latency of the smaller operations by 42% until a frequency of 8 `gets` per yield. Yields add some overhead, and yielding more frequently pushes the store to saturation in this case. As a result, all requests see increased response times. Extensions should yield frequently, but yielding too often is wasteful. Splinter may be able to help with this in the future; Splinter could provide extensions with a yield that is ignored if called too quickly in succession, avoiding the full yield cost.

Figure 11 shows how uncooperative extensions impact system performance. Here, the client invoked a small fraction of extension operations that executed an infinite loop. The remaining fraction of requests invoked a small extension that performed a single `get`. Splinter performs well in the presence of misbehaving extensions. Throughput is steady at 3 Mops/s irrespective of the fraction of misbehaving requests. Median latency isn't shown, but it is steady as well. Tail latency suffers as more requests misbehave, though it is within 100 µs for fractions as high as one in a million requests.

Note that one in a million requests (1e-6) is harsh. The store can execute more than 4 Mop/s, so this represents a misbehaving invocation starting every quarter second; at 1e-5 misbehavior starts about once every 25 ms.

5.5 Aggregation Extension

Online data aggregation is a common task for applications. For example, a user might send a query demanding a movie studio's total earnings in the year 2017. With a key-value data model, this would require two round-trips to storage: one to fetch the list of movies made by the studio and one to fetch the box-office earnings of each of the movies. Splinter improves the user-facing and server-side performance of these types of queries by allowing applications to inexpensively embed their data model (studios and movies) and operations (total earnings aggregation) within storage.

Figure 12 compares a completely client-based and a Splinter extension-based implementation of such an aggregation over 4 records. Each of the store's 1,024 tenants owned a table with 300 K indirection lists pointing to 1.2 million records, totalling about 100 GB of stored data.

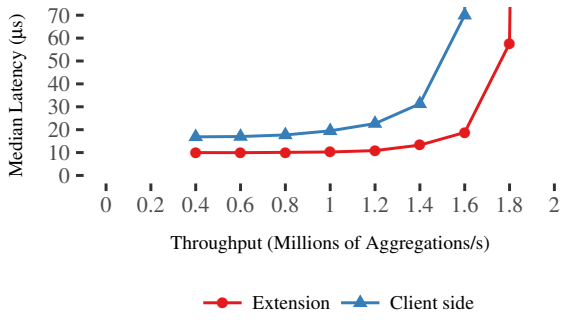


Figure 12: Aggregation throughput versus latency. Aggregations combine 4 records. Under low load, the median latency of a client-side implementation is 1.6× that of an extension-based implementation. Using an extension also improves saturating throughput from 1.2 M to 1.6 M aggregations per second.

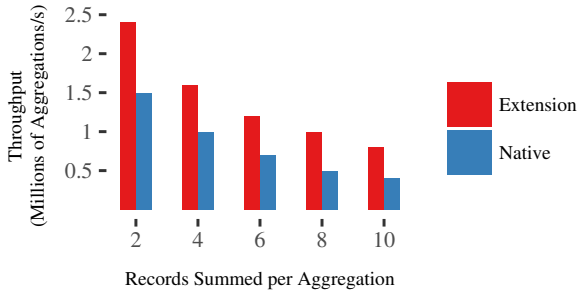


Figure 13: Saturating throughput of aggregation versus the number of aggregated records. The extension-based implementation outperforms the client-side implementation irrespective of the number of records aggregated. The gains are highest when aggregations are over two records (2.4 M versus 1.5 M aggregations per second).

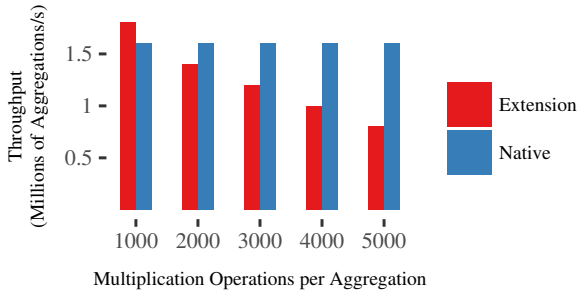


Figure 14: Saturating throughput of the aggregation extension versus the amount of compute per aggregation. After aggregating 2 records, each operation raised the result to the power n, implemented as n 64-bit multiplications (hence the x-axis). Increasing the order (n) increases server-side compute in the extension-based implementation, hurting throughput. At an order of 5000, the client-side approach is 2× faster.

The client-based implementation first performed a `get()` to retrieve an indirection list followed by a `multiget()` (a single RPC requesting values for multiple keys) to fetch all of the records indicated in the indirection list. The first field from each of the returned objects is summed up into a single 64-bit result. The extension-based implementation invoked a Splinter extension called `aggregate()` with the same functionality as the client-based approach.

Pushing the aggregation from the client to the server has two key benefits. First, it improves performance from the client's perspective: the extension-based implementation reduces median latency by 38% (from 16 µs to 10 µs) under low load with larger gains under higher loads. This improvement is mainly due to a reduction in the number of round-trips; unlike the client-based extension, the `aggregate()` extension doesn't need to wait for the store to return an indirection list before it can start aggregation. Second, it improves performance from the server's perspective as well. Splinter's extension invocations are more expensive than plain `get()` operations (§5.2), but they eliminate some of the costly network and RPC processing. Hence, saturating throughput improves from 1.2 M to 1.6 M aggregations per second.

Note, this improvement comes in a challenging case for Splinter; at 40 Gbps, Splinter is never network limited. These results show that even if a store is CPU-limited, pushing compute to the store can still provide a throughput benefit, since it can mitigate request processing overheads. On slower networks, Splinter would provide more of a benefit since extensions can reduce network load.

Figure 13 shows the impact of the number of records aggregated on the saturating throughput of the extension-based and client-based implementation. In both approaches, increasing the number of records aggregated increases the work the store has to do per request (`aggregate()`/`multiget()`), and, hence, decreases the overall throughput of the system. However, if that work is simple (like summation) it is always better to aggregate at the store. The gain in saturating throughput of the extension-based aggregation is always more than 50%.

For compute-intensive operations, the extra CPU cost of running extensions at the store can outweigh the gains of fewer RPCs. Figure 14 explores this effect. After adding the first field of two records, each operation raises the result to the power n (with n 64-bit multiplications). Using an extension, increasing n above 2,000 slows the store and decreases saturating throughput from 1.8 M to 800 K aggregations per second. The client-side approach can hold throughput constant at 1.6 M aggregations per second; the client has enough idle CPU capacity to compute the result. This shows that extensions are ideal for operations with modest amounts of compute. For compute-intensive operations over data stored on high-load servers, clients should fetch data and perform operations locally.

5.6 TAO Extension

TAO [10] is a graph-oriented in-memory cache used at Facebook to hold objects from the social graph and as-

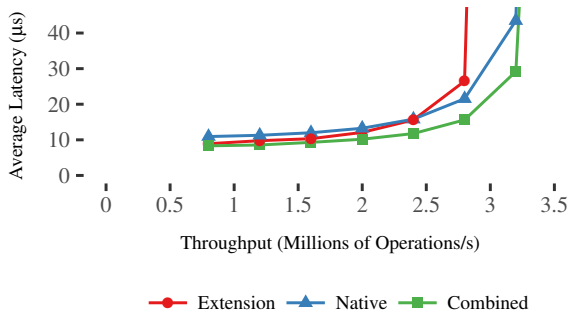


Figure 15: TAO extension throughput versus latency. With 60% `object_get` and 40% `assoc_range` operations, the TAO extension can reach 2.8 Mop/s before saturating with an average latency of 30 µs. By using native `get()` operations for `object_get`, the extension-based approach can outperform a purely client-side implementation by 400 Kop/s.

sociations between those objects. TAO is well-suited to Splinter. It is designed for interactive data, but it embeds knowledge about Facebook's workload to decrease round-trips to the store, which eliminates client-side stalls and improves server-side efficiency. We have implemented its simple operations as an 800-line Splinter extension.

Full details of TAO are beyond the scope of this paper, but the basics are simple. Aside from object put/get, TAO's *association lists* (e.g. `user1`'s "likes") allow one object to be associated to another via a typed, directed edges. For example, `user1`'s "likes" may be represented as an association list `(user1, likes)` → [`post1`, `post32`]. Association lists provide simple operations for adding, removing, and counting associations. Entries in association lists are timestamped, and range operations over association lists to fetch subsets of them are common ("get the first 10 entries in the `(user1, likes)` association list").

Figure 15 shows Splinter's performance under three different configurations: an extension-based approach (Extension), a client-based approach (Native), and a combined approach (Combined) that implemented `object_get` using native `get()` operations, and `assoc_range` using an extension. The workload was configured to issue a mix of 60% `object_get` and 40% `assoc_range` operations. We picked this ratio based on Facebook's reported TAO workload [10], which is dominated by reads (99.8%) mostly from these two operations. Each of the 1,024 tenants on the storage node owned a graph with half a million objects and two million edges (associations), totalling about 100 GB of stored data.

Since a significant fraction of requests are single round-trip `object_gets`, the client-based approach has a better saturating throughput than the extension-based approach. However, combining the two improves saturating throughput from 2.8 Mop/s to 3.2 Mop/s at a latency of 31 µs; the native `get()` helps eliminate the isolation overhead while executing an `object_get`, and the extension helps reduce the number of round-trips required by an `assoc_range`.

This makes Splinter competitive with FaRM's TAO implementation which is the fastest known implementation. Interestingly FaRM, takes the opposite approach of Splinter. On FaRM, TAO operations use multiple RDMA reads and careful object layout. FaRM reported 6.3 Mops/s (about 200 Kop/s/core) with a 41 µs average latency; Splinter performs about 400 Kops/s/core with lower latency. Differences in hardware and experimental setup likely account for some of the differences, but it shows Splinter's CPU-active server approach is competitive against FaRM's CPU-passive server approach. Furthermore, Splinter maintains a simple, remote procedure call interface, and the TAO extension enforces strong abstract data types. Splinter TAO clients have no knowledge of the internal layout of the stored data objects.

6 Related Work

Shipping computation to data and isolating untrusted code are well-studied, and Splinter builds on prior work. However, prior work does not address multi-tenancy at Splinter's granularity and number of tenants; further, no work addresses these issues with its throughput and latency goals, which are far beyond most cloud storage systems.

Low-latency RDMA-based Storage Systems. Low-latency, high-throughput key-value stores are now thousands of times faster than conventional cloud storage by exploiting RDMA, kernel-bypass, and DRAM [22, 23, 36, 44, 45, 55]. These systems are well-understood for small, regular workloads, but their simple (get/put, read/write) interfaces make them easy to optimize internally at the expense of application efficiency, since they force clients to make many round trips to storage and to compute locally [21]. RDMA lowers CPU overhead for transmit, but it cannot make up for the fundamental inefficiency of moving large amounts of data over the wire; receivers must still perform the same computation on the data that a server could have. Splinter eliminates this waste, while still using efficient kernel-bypass networking. At 40 Gbps a Splinter store is never network bound, so combining Splinter's approach with (one- or two-sided) RDMA verbs could provide a benefit by freeing up additional compute on store servers.

6.1 Pushing Computation to Storage

MapReduce [18] and Spark [73] ship code to data sets, though latency is not a concern. Even when compute is shipped to a storage (HDFS [63]) node, data is still copied via interprocess communication. Untrusted extensions, like those in Splinter, could eliminate these overheads.

Some distributed systems and frameworks support composing internal storage abstractions to synthesize new services [3, 4, 11, 28, 48, 62]. Malacology [62] claims stor-

age extensions have been popular in the Ceph distributed file system, showing that extensions are useful to developers. In these systems, extensions are trusted, so they don't work for cloud storage; Splinter is also focused on tight integration of fine-grained computation and storage rather than on coarse composition of software services. Comet [26] embedded sandboxed Lua extensions into a decentralized hash table to allow application-specific extensions to get/put behavior. Lua's entry/exit costs are low; it is unclear how the performance of its just-in-time (JIT) compiled runtime would compare to Splinter.

SQL. SQL may be the most widely used approach to ship computation to data, and it also supports use as a stored procedure language [50, 54]. In-memory databases have placed pressure on performance, resulting in JIT compilation for SQL [25, 53]. With JIT, queries run fast, and calls back-and-forth between the database and user logic are inexpensive. SQL is type safe, so it is also easy to isolate. SQL's main drawback is that it is declarative. Often, this is a benefit, since it can use runtime information for optimization, but this also limits its generality. Implementing new functionality, new operators, or complex algorithms in SQL is difficult and inefficient. Some have extended SQL for specific domains, like graph processing [52], scientific computing [47, 57] and simulation [12], showing that SQL by itself is insufficient for many domains.

Native-code Extensions. The popular Redis [1] in-memory store supports native extensions. In FaRM [22, 23], an RDMA-based in-memory store, applications are written as native, storage-embedded functions that are statically compiled into the server. These systems don't allow extensions to be loaded at runtime, and application code is trusted so it does not work for multi-tenant cloud storage. Similarly, H-Store [39], VoltDB [65], and Hazelcast [29] are in-memory stores that support Java-based procedures, though none of them provide multi-tenancy.

6.2 Fault Isolation

Software-fault isolation (SFI) sandboxes untrusted code within a process (or OS kernel [33, 61, 67]) with low control transfer costs [9, 24, 27, 49, 72]. Both hardware isolation [66] and SFI [69] were applied to Postgres [64], which pioneered database extensions [68]. SFI still requires protected data to be copied in/out of extensions, since it relies on hardware paging or address masking that can only restrict access to contiguous memory regions.

Language-level approaches to kernel extension [8, 30] closely match Splinter's design and goals. SPIN let language-isolated extensions run as part of the kernel. It eliminated runtime overheads (aside from garbage collection), since extensions were compiled; it eliminated control transfer overheads, since it didn't require page table switching; and it eliminated copying between protection domains, since type-safe pointers worked as capabilities. Like Splinter, where tenants must write Rust code, a key downside of SPIN was that extensions had to be written in Modula-3, not C, so legacy code couldn't be used. Java also "sandboxed" applets using type-safety and specialized class loaders, which supported inexpensive control transfer and data access between domains [70].

Using Rust for low-cost, zero-copy isolation has been used for inexpensive software fault isolation both generally [5] and for network packet processing pipelines [56]. Splinter builds on these ideas, bringing them to storage and moving beyond static domains to a runtime extensible service. Tock [43] is an embedded OS that decomposes its kernel into untrusted *capsules* by exploiting Rust's safety. Tock's capsules are similar to Splinter's extensions, but they don't protect against denial of service (infinite loops) and capsules are static – they can't be added to a running kernel. These also differ from Splinter in that they assume a small number of trust domains; they are targeted at software decomposition. Splinter targets dense multi-tenancy with no static bound on the number of trust domains.

7 Conclusion

In-memory storage can significantly accelerate data-intensive applications, including those that need fine-grained and real-time access to data. However, as Dennard scaling ends, future cloud storage must not only be faster but also more efficient. Splinter shows that soon legacy hardware isolation techniques will limit resource provisioning granularity in the cloud, but it also provides a way forward. Systems must evolve to support granular, low-overhead shipping of compute to storage, and lightweight isolation between small compute tasks. Splinter works toward that evolution by discarding hardware isolation in favor of static safety checks. As a result, it supports thousands of tenants that can all access data in tens of microseconds while customizing storage operations to their needs and while performing millions of remote operations on modern multicore machines.

Acknowledgments

Thanks to Ankit Bhardwaj and Ethan Ransom for contributing to Splinter; to Abhiram Balasubramanian, Anton Burtsev, and Amit Levy for the conversations that helped lead us to this work; to the reviewers for their comments; and to our shepherd, Jon Howell. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1750558 and CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Facebook and VMware.

References

[1] Redis. http://redis.io/. Accessed: 2018-09-27.

[2] The Rust Programming Language. http://www.rust-lang.org/en-US/. Accessed: 2018-09-27.

[3] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBLER, T., WEI, M., AND DAVIS, J. D. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI '12, USENIX Association, pp. 1–14.

[4] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed Data Structures Over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, PA, 2013), SOSP '13, ACM, pp. 325–340.

[5] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z., AND RYZHYK, L. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, 2017), HotOS '17, ACM, pp. 156–161.

[6] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the Killer Microseconds. *Communications of the ACM 60*, 4 (Mar. 2017), 48–54.

[7] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI '14, USENIX Association, pp. 49–65.

[8] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, ACM, pp. 267–283.

[9] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, 2008), NSDI '08, USENIX Association, pp. 309–322.

[10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC '13, USENIX Association, pp. 49–60.

[11] BROWN, A., OPPENHEIMER, D., KEETON, K., THOMAS, R., KUBIATOWICZ, J., AND PATTERSON, D. A. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the The 7th Workshop on Hot Topics in Operating Systems* (Washington, DC, 1999), HotOS '99, IEEE Computer Society, pp. 32–37.

[12] CAI, Z., VAGENA, Z., PEREZ, L., ARUMUGAM, S., HAAS, P. J., AND JERMAINE, C. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 637–648.

[13] CARRUTH, C. [SLH] Introduce a new pass to do Speculative Load Hardening to mitigate. http://reviews.llvm.org/rL336990, 2018. Accessed: 2018-09-27.

[14] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (June 2008), 4:1–4:26.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, 2010), SoCC '10, ACM, pp. 143–154.

[16] CORBET, J. KAISER: hiding the kernel from user space. http://lwn.net/Articles/738975/. Accessed: 2018-09-27.

[17] CORBET, J. Meltdown/Spectre mitigation for 4.15 and beyond. http://lwn.net/Articles/744287/, 2018. Accessed: 2018-09-27.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation* (Berkeley, CA, 2004), OSDI '04, USENIX Association, pp. 10–10.

[19] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, 2007), SOSP '07, ACM, pp. 205–220.

[20] DPDK PROJECT. Data Plane Development Kit. http://dpdk.org/. Accessed: 2018-09-27.

[21] DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. RDMA Reads: To Use or Not to Use? *IEEE Data Engineering Bulletin 40*, 1 (2017), 3–14.

[22] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, 2014), NSDI '14, USENIX Association, pp. 401–414.

[23] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, 2015), SOSP '15, ACM, pp. 54–70.

[24] FORD, B., AND COX, R. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Berkeley, CA, 2008), USENIX ATC '08, USENIX Association, pp. 293–306.

[25] FREEDMAN, C., ISMERT, E., AND LARSON, P. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin 37*, 1 (2014), 22–30.

[26] GEAMBASU, R., LEVY, A. A., KOHNO, T., KRISHNAMURTHY, A., AND LEVY, H. M. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, BC, 2010), OSDI '10, USENIX Association, pp. 323–336.

[27] GOOGLE LLC. NaCl and PNaCl. http://developer.chrome.com/native-client/nacl-and-pnacl. Accessed: 2018-09-27.

[28] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation* (Berkeley, CA, 2000), OSDI '00, USENIX Association.

[29] HAZELCAST. Hazelcast the Leading In-Memory Data Grid - Hazelcast.com. http://hazelcast.com/. Accessed: 2018-09-27.

[30] HUNT, G., AND LARUS, J. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review 41/2* (April 2007), 37–49.

[31] INTEL CORPORATION. Flow Director. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. Accessed: 2018-09-27.

[32] INTEL CORPORATION. Intel®Data Direct I/O technology. http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. Accessed: 2018-09-27.

[33] JACOBSEN, C., KHOLE, M., SPALL, S., BAUER, S., AND BURTSEV, A. Lightweight Capability Domains: Towards Decomposing the Linux Kernel. *SIGOPS Operating Systems Review 49*, 2 (Jan. 2016), 44–50.

[34] JUNG, R. LLVM loop optimization can make safe programs crash #28728. http://github.com/rust-lang/rust/issues/28728, 2018. Accessed: 2018-09-27.

[35] JUNG, R., JOURDAN, J., KREBBERS, R., AND DREYER, D. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages 2* (2018), 66:1–66:34.

[36] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, 2014), SIGCOMM '14, ACM, pp. 295–306.

[37] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI '16, USENIX Association, pp. 185–201.

[38] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter RPCs can be General and Fast. *CoRR abs/1806.00680* (2018).

[39] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1496–1499.

[40] KIRIANSKY, V., AND WALDSPURGER, C. Speculative Buffer Overflows: Attacks and Defenses. *CoRR abs/1807.03757* (2018).

[41] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (2019), S&P '19.

[42] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), CGO '04, IEEE, pp. 75–86.

[43] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, 2017), SOSP '17, ACM, pp. 234–251.

[44] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (New York, NY, 2015), ISCA '15, ACM, pp. 476–488.

[45] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, 2014), NSDI '14, USENIX Association, pp. 429–444.

[46] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 973–990.

[47] MAAS, R., HYRKAS, J., TELFORD, O., BALAZINSKA, M., CONNOLLY, A., AND HOWE, B. Gaussian Mixture Models Use-Case: In-Memory Analysis with Myria. *Third International Workshop on In-Memory Data Management and Analytics (IMDM'15)* (2015).

[48] MACCORMICK, J., MURPHY, N., NAJORK, M., THETH, C. A., AND ZHOU, L. Boxwood: Abstractions As the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design and Implementation* (Berkeley, CA, 2004), vol. 6 of *OSDI '04*, USENIX Association, pp. 8–8.

[49] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, 2006), USENIX-SS '06, USENIX Association.

[50] MICROSOFT, INC. Transact-SQL Reference (Database Engine). http://docs.microsoft.com/en-us/sql/t-sql/language-reference. Accessed: 2018-09-27.

[51] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference* (Santa Clara, CA, July 2015), USENIX ATC '15, USENIX Association, pp. 291–305.

[52] NEO4J, INC. Neo4j, the World's Leading Graph Database. http://neo4j.com/. Accessed: 2018-09-27.

[53] NEUMANN, T. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment 4*, 9 (2011), 539–550.

[54] ORACLE, INC. Oracle Database 12c PL/SQL. http://www.oracle.com/technetwork/database/features/plsql/index.html. Accessed: 2018-09-27.

[55] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (Aug. 2015), 7:1–7:55.

[56] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI '16, USENIX Association, pp. 203–216.

[57] PARADIGM4, INC. Paradigm4: Cretators of SciDB a computational Database. http://www.paradigm4.com/.

[58] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), SOSP '17, ACM, pp. 325–341.

[59] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, 2018), OSDI '2018, USENIX Association.

[60] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[61] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), OSDI '96, USENIX Association, pp. 213–227.

[62] SEVILLA, M. A., WATKINS, N., JIMENEZ, I., ALVARO, P., FINKELSTEIN, S., LEFEVRE, J., AND MALTZAHN, C. Malacology: A Programmable Storage System. In *Proceedings of the 12th European Conference on Computer Systems* (2017), Eurosys '17, ACM, pp. 175–190.

[63] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), IEEE, pp. 1–10.

[64] STONEBRAKER, M., AND KEMNITZ, G. The POSTGRES Next Generation Database Management System. *Communications of the ACM 34*, 10 (Oct. 1991), 78–92.

[65] STONEBRAKER, M., AND WEISBERG, A. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin 36*, 2 (2013), 21–27.

[66] SULLIVAN, M., AND STONEBRAKER, M. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the VLDB Endowment* (1991), VLDB '91, VLDB Endowment, pp. 171–180.

[67] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 207–222.

[68] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL: Documentation: 10: H.4. Extensions. `http://www.postgresql.org/docs/10/static/external-extensions.html`. Accessed: 2018-09-27.

[69] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles* (New York, NY, 1993), SOSP '93, ACM, pp. 203–216.

[70] WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. Extensible Security Architectures for Java. *SIGOPS Operating Systems Review 31*, 5 (Oct. 1997), 116–128.

[71] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, 2015), SOSP '15, ACM, pp. 87–104.

[72] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), S&P '09, IEEE, pp. 79–93.

[73] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI '12, USENIX Association, pp. 15–28.

Neural Adaptive Content-aware Internet Video Delivery

Hyunho Yeo Youngmok Jung Jaehong Kim Jinwoo Shin Dongsu Han
KAIST

Abstract

Internet video streaming has experienced tremendous growth over the last few decades. However, the quality of existing video delivery critically depends on the bandwidth resource. Consequently, user quality of experience (QoE) suffers inevitably when network conditions become unfavorable. We present a new video delivery framework that utilizes client computation and recent advances in deep neural networks (DNNs) to reduce the dependency for delivering high-quality video. The use of DNNs enables us to enhance the video quality independent to the available bandwidth. We design a practical system that addresses several challenges, such as client heterogeneity, interaction with bitrate adaptation, and DNN transfer, in enabling the idea. Our evaluation using 3G and broadband network traces shows the proposed system outperforms the current state of the art, enhancing the average QoE by 43.08% using the same bandwidth budget or saving 17.13% of bandwidth while providing the same user QoE.

1 Introduction

Internet video has experienced tremendous growth over the last few decades. Recent market reports indicate people around the world watch 5.75 hours of online video per week on average [10] and video traffic is expected to quadruple in the next five years [26, 63]. Current video delivery infrastructure has been successful in handling the scalability challenges with two key technologies. First, at the server side, distributed computing technologies enabled content delivery at Internet scale. Second, at the client side, adaptive bitrate (ABR) streaming addressed the problem of bandwidth heterogeneity and its variations across time and space. Techniques at both ends evolved over time to optimize user quality of experience (QoE) as it ultimately impacts the revenue of various stakeholders [22, 27, 77].

However, the limitation of existing content distribution networks (CDNs) is that its quality heavily depends on the bandwidth between servers and clients. When the bandwidth resource becomes scarce, user QoE suffers directly [43, 47]. Bitrate adaptation has been the primary tool to relax the problem [52]. Nevertheless, its sole reliance on network resource is a fundamental limitation.

Inspired by the ever-increasing clients' computational power and recent advances in deep learning, this paper identifies an alternative and complementary approach to enhancing the video quality. We apply a deep neural network (DNN)-based quality enhancement on video *content* utilizing the *client computation* to maximize user QoE. In particular, a deep learning model learns a mapping from a low-quality video to a high-quality version, e.g., super-resolution. This enables clients to obtain high-definition (e.g., 1080p) video from lower quality transmissions, providing a powerful mechanism for QoE maximization on top of bitrate adaption.

Leveraging client computation via DNNs impacts the server/client system and introduces a number of non-trivial challenges:

- First, the CDN servers have to provide a DNN model for the content they provide. However, it is difficult to guarantee the test performance of DNN's predictions. It is especially unreliable for unseen/new content, presenting a significant barrier to deployment.
- Second, client devices are heterogeneous. Their computational power varies widely and may even exhibit temporal variation due to multiplexing. Nevertheless, DNN-based quality enhancement must occur at real-time to support online video streaming.
- Finally, the DNN-based quality enhancement has a cascading effect on ABR-based QoE optimization. The quality now depends on the availability of DNNs at the client in addition to the available bandwidth. Thus,

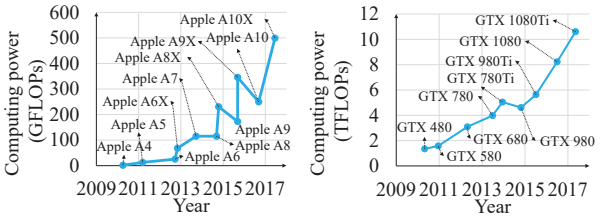
existing ABR algorithms must reflect the changes.

This paper presents NAS, the first video delivery framework that applies DNNs on video content using client's computational power to maximize user QoE. We present a system design that runs on top of Dynamic Adaptive Streaming over HTTP (DASH) framework. NAS addresses the challenges by introducing new system designs. To guarantee reliable quality enhancement powered by DNN, it takes a content-aware approach in which a DNN is trained for each content separately. The idea is to leverage the DNN's overfitting property and use the training accuracy to deliver predictable high performance, instead of relying on the unpredictable test accuracy. Next, to meet the real-time constraints on heterogeneous environments, we use multiple scalable DNNs that provide anytime prediction [24, 36]. Such DNN architectures can adaptively control their computational cost given resource budget. NAS clients choose a DNN (from multiple options) that best fits their resources and adapt to temporal variations in computing power at each time epoch. The scalable DNN also enables the use of a partially downloaded DNN, bringing an incremental benefit in downloading a DNN model. Finally, to reconcile the ABR-based QoE optimization and DNN-based quality enhancement, we devise a content enhancement-aware ABR algorithm for QoE optimization. To this end, we integrate our design into the state-of-the-art ABR algorithm [52] that uses reinforcement learning [68]. The algorithm decides when to download a DNN model and which video bitrate to use for each video chunk.

We evaluate NAS using a full system implementation. Our evaluation on 27 real videos and 17.8 hours of real-world network traces [8] using six different GPU models shows NAS delivers substantial benefit in a wide range of settings and is able to meet the real-time constraint on desktop class GPUs of varying capacity. In particular, it improves user QoE between 63.80-136.58% compared to BOLA [66] used in DASH [4] and between 21.89-76.04% compared to Pensieve, the state-of-the-art ABR design. Finally, we provide in-depth performance analysis of individual system components.

In summary, we make three key contributions:

- **End-to-end video delivery system:** NAS is an end-to-end video streaming system that integrates the content-aware approach, DNNs for super-resolution, scalable anytime prediction, and mechanisms for handling device heterogeneity on top of an existing adaptive streaming framework.
- **Use of DNNs in adaptive streaming:** NAS is the first system to apply super-resolution DNNs over video content in the context of adaptive streaming. From the machine learning (ML) side, we are the first to apply DNN-streaming, super-resolution, and anytime prediction to adaptive streaming.
- **Content-aware DNN:** NAS streams video along with the corresponding content-aware DNN to its clients. This is a key enabler and a novel component of NAS, which can be also viewed as a new approach to video coding.



(a) Mobile (b) Desktop

Figure 1: Growth of GPU's processing power

2 Motivation and Goal

Traditional approaches to improving video stream quality include: using better codecs [11, 12]; optimizing adaptive bitrate algorithms [20, 39, 42]; choosing better servers and CDNs [17, 50, 74]; and using coordination among clients and servers through a centralized control plane [51, 54]. These approaches focus on how to best utilize the network resource, but suffer from two common limitations.

Under-utilization of client's computation. Market reports [10, 57] indicate the majority of users watch video primarily on PCs, which have significant computation power. Mobile devices, which is the next popular platform, are also equipped with power-efficient graphic processing units (GPUs) [29]. Figure 1 shows the exponential growth in GPU's computing power over time on mobile devices and desktop PCs. Latest mobile devices even have dedicated hardware for neural processing [7]. However, the current video delivery infrastructure *under-utilizes* client's computational power. With their growing computational capacity and ever-increasing demand for bandwidth, we envision a video delivery system in which clients take an active role in improving the video quality.

Limitation of current video coding. Video episodes often contain redundancy that occurs at large timescales. For example, consider a popular sports game (e.g., NBA finals) watched by millions of people. Same objects (e.g., balls and players) and scenes (e.g., basketball court) show up repeatedly. Similarly, redundancy is also found within episodes of a TV show, games in a sports league, and videos from the same streamers. Such frequently reoccurring high-level features contain valuable information that can be leveraged for video coding. However, standard

video coding, such as MPEG and H.26x, only captures spacial and short-term redundancy, lacking any mechanisms to exploit motion picture's high-level features. Within a group of pictures (GOP), inter-frame coding encodes the difference between adjacent frames to compress a motion picture [30]. However, a GOP is typically on the order of seconds for online video [13], making it impossible to capture redundancy that occurs at large timescales. As long as codecs compress video only within a GOP (arguably a fundamental constraint for streaming), using sophisticated codecs would not completely close this gap.

Motivated by this, we envision a video delivery system that exploits such redundancy by capturing the high-level features and applies additional client computation to augment the limitation of traditional video encoding. To this end, we utilize DNNs that abstract meaningful features from a low-level representation of data [23].

System goal. Our goal is to design a practical system that augments the existing infrastructure to optimize user QoE. As the first step, we consider servicing on-demand videos, as opposed to live streams, and using personal computers that have desktop-class GPUs. We propose a redesign of the video delivery infrastructure to take advantage of client computation to a greater degree. For quality enhancement, we utilize super-resolution that takes low-quality video as input and generates an "up-scaled" version. We choose super-resolution because significant advances have been made recently [28, 45, 49]. While we scope our study to desktop-class GPUs and super-resolution, we believe the framework is generic enough to accommodate different types of DNN models and devices.

3 Background and Related Work

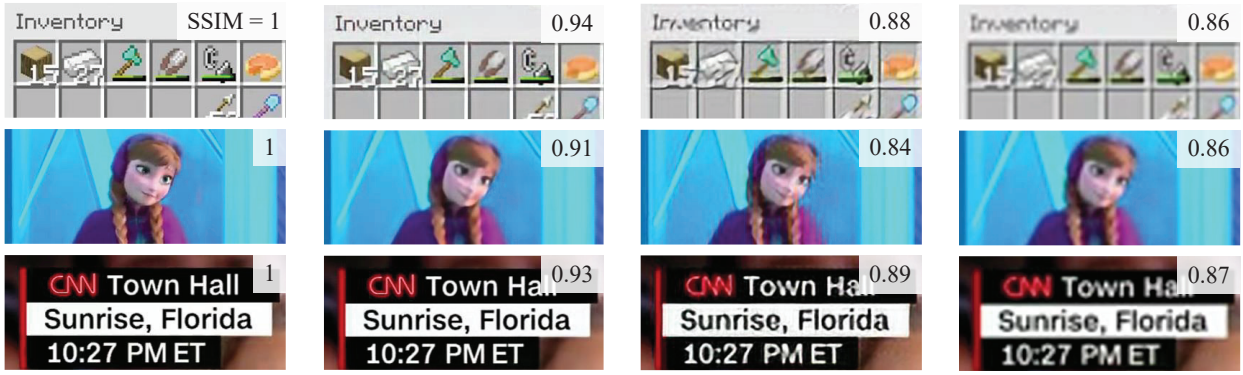
Adaptive streaming (e.g., Apples HLS [1], DASH [2]) is designed to handle unpredictable bandwidth variations in the real world. Video is encoded into various bitrates (or resolutions) and divided into fixed length chunks, typically 2 – 10 seconds. An adaptive birate algorithm (ABR) decides the bitrate for each video chunk. Traditional ABR algorithms select bitrates using heuristics based on the estimated network bandwidth [42] and/or the current size of the client-side playback buffer [66]. MPC [77] and Pensieve [52] demonstrate that directly optimizing for the desired QoE objective delivers better outcomes than heuristics-based approaches. In particular, Pensieve uses deep reinforcement learning and learns through "observations" how past decisions and the current state impact the video quality. Oboe [21] dynamically adjusts the ABR parameters depending on the network conditions consulting the offline pre-computation result. Although these algorithms successfully cope with bandwidth variations, they consider neither the effect of client-side quality enhancement nor the dynamics of simultaneously streaming a DNN and video chunks.

Super-resolution recovers a high-resolution image from a single or multiple *lower resolution* image(s). Super-resolution has been used in a variety of computer vision applications, including surveillance [78] and medical imaging [65], where the original high-quality image/video is not available. Recent studies use DNNs [28, 45, 49] to learn low-resolution to high-resolution mapping and demonstrate a significant performance gain over non-DNN approaches [25, 64]. In particular, MDSR [49] is a state-of-the-art DNN that integrates the residual neural network architecture [34] and supports multi-scale inputs. In NAS, we apply super-resolution on top of adaptive streaming to improve user QoE by enhancing low-quality videos at the client side.

Scalable DNN is an emerging type of DNN designed to dynamically adapt to computational resource constraints, enabling *anytime prediction* [36]. A shallow and a deep network are used in resource-constrained and -sufficient environments respectively [24, 36]. ISResNeXt [48] alternatively uses a thin and a wide network that adapts to the width (or channels) of a DNN. Scalable DNN has been applied primarily to image classification/detection tasks. NAS applies anytime prediction to super-resolution and uses it delivering incremental quality enhancement in a streaming context.

DNN-based media compression. Recent studies [18, 61, 71] have shown DNN-based image compression outperforms traditional image codecs, such as JPEG2000 and WebP. The benefit over conventional codecs comes mainly from two aspects: 1) directly optimizing for the target quality metric and 2) adapting the codec configuration based on the image rather than using a fixed configuration [61]. Applying this to video, however, involves significant challenges including the problem of reducing inter-frame redundancy across DNN-encoded images. A recent work [72] performs both I-frame compression and frame interpolation using DNNs. However, the DNN-based video compression is still at its early stage and only offers "comparable performance to MPEG-2" and falls short in delivering real-time decoding [72]. NAS aims to augment existing video delivery using DNN—it applies super-resolution DNNs on top of traditional video codecs by applying quality enhancements frame-by-frame.

Video processing systems. Back-end video processing systems have been of growing importance due to the scale required for video encoding. Studies have reported that latency for fast interactive sharing, system efficiency in encoding, scalability and fault tolerance are major issues [31,37,70]. SVE [37] presents a backend system for video



(a) Original (1080p) (b) Content-aware DNN (c) Content-agnostic DNN (d) 240p

Figure 2: 240p to 1080p super-resolution results
(Content type – 1st row: Game [15], 2nd row: Entertainment [14], 3rd row: News [16])

processing used in Facebook. ExCamera [31] uses massive parallelism to enable interactive and collaborative editing. They focus on solving distributed system problems within a datacenter without changing the clients, whereas we focus on the division of work between the servers and clients.

Studies on video control plane [32, 41, 44, 51] identify spatial and temporal diversity of CDNs in performance and advocate for an Internet-scale control plane which coordinates client behaviors to collectively optimize user QoE. Although they control client behaviors, they do not utilize client computation to directly enhance the video quality.

4 Key Design Choices

Achieving our goal requires redesigning major components of video delivery. This section describes the key design choices we make to overcome practical challenges.

4.1 Content-aware DNN

Key challenge. Developing a universal DNN model that works well across all Internet video is impractical because the number of video episodes is almost infinite. A single DNN of finite capacity, in principle, may not be expressive enough to capture all of them. Note, a fundamental trade-off exists between generalization and specialization for any machine learning approach (i.e., as the model coverage becomes larger, its performance degrades), which is referred to as the 'no free lunch' theorem [75]. Even worse, one can generate 'adversarial' new videos of arbitrarily low quality, given any existing DNN model [38, 58], making the service vulnerable to reduction of quality attacks.

NAS' content-aware model. To tackle the challenge, we consider a content-aware DNN model in which we use a

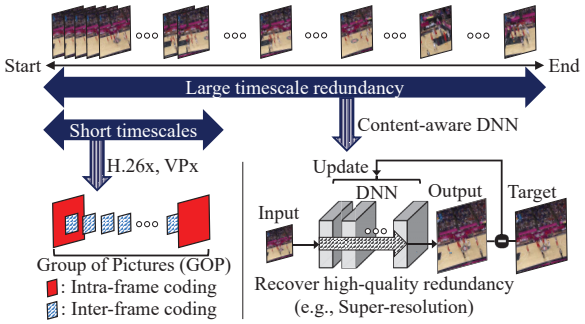


Figure 3: Content-aware DNN based video encoding

different DNN for each video episode. This is attractive because DNNs typically achieve near-zero training error, but the testing error is often much higher (i.e., over-fitting occurs) [67]. Although the deep learning community has made extensive efforts to reduce the gap [40, 67], relying on the DNN's testing accuracy may result in unpredictable performance [38, 58]. NAS exploits DNN's inherent overfitting property to guarantee reliable and superior performance.

Figure 2 shows the super-resolution results of our content-aware DNN and a content-agnostic DNN trained on standard benchmark images (NTIRE 2017 dataset [19]). We use 240p images as input (d) to the super-resolution DNNs to produce output (b) or (c). The images are snapshots of video clips from YouTube. The generic, universal model fails to achieve high quality consistently over a variety of contents—in certain cases, the quality degrades after processing. In §5.1, we show how to design a content-aware DNN for adaptive streaming.

The content-aware approach can be seen as a type of video compression as illustrated in Figure 3. The content-aware DNN captures redundancy that occurs at large time scales (e.g. multiple GOPs) and operates over the entire video. In contrast, the conventional codecs deals with re-

Model name	Compute capacity (Single precision)	Price
GTX 1050 Ti	1.98 TFLOPS	$139
GTX 1060	3.86 TFLOPS	$249
GTX 1070	5.78 TFLOPS	$379
GTX 1070 Ti	7.82 TFLOPS	$449
GTX 1080	8.23 TFLOPS	$559
GTX 1080 Ti	10.61 TFLOPS	$669
Titan Xp	10.79 TFLOPS	$1,200

Table 1: Nvidia's desktop GPU (Geforce 10 series)

dundancy within a frame or between frames within a GOP. In NAS, we demonstrate the new encoding scheme using per-video super-resolution DNNs. However, we believe the content-aware approach can be applied to a series of videos (of similar content) and extended to work with different types of DNNs, such as frame interpolation [56], as discussed in our position paper [76].

4.2 Multiple, Scalable DNNs

Key challenge. The available capacity of computing changes across time and space because of heterogeneity of client devices, changes in workloads, and multiplexing. Table 1 shows even within the desktop-class GPUs the computational power varies up to 5.68 times. Nevertheless, real-time inference is required for online video streaming—the DNN inference has to be at least as fast as the playback rate. However, existing super-resolution DNNs [28, 45, 49] require a fixed amount of computing power and cannot adapt to time-varying capacity. Thus, using a single DNN either under-utilizes client's GPU or does not meet the real-time requirement.

NAS' multiple, scalable DNN design. To tackle the challenge, NAS offers multiple DNNs and let clients dynamically choose one that fits their resource. Similar to multiple bitrates that adaptive streaming offers, we provide a range of DNN options that differ in their inference time (or computational requirements). NAS servers provide multiple DNN specifications as part of the video manifest file. We provide a light-weight mechanism that does not require clients to download the DNNs for choosing the right DNN from available options.

However, using multiple DNNs introduces another challenge. Because the size of DNN grows proportional to its computation requirement, DNNs designed for high-end GPU devices can be very large (a few MBs). It can take a long time to download and utilize the DNN. To address the issue, we design a scalable DNN that enables a client to utilize a partially downloaded model in an incremental fashion. The scalable DNN consists of multiple bypass-able intermediate layers, enabling a partial DNN without the intermediate layers to generate the output as shown in Figure 4. In addition, the design naturally accommodates temporal variation in computational power due to multiplexing. When the computational resource is abundant, clients can use all layers, otherwise they can opportunistically bypass any number of intermediate layers, enabling *anytime prediction* [24, 36, 48]. Finally, the use of multiple scalable DNNs allows each device to benefit from partially downloaded DNNs and provides the same level of temporal adaptation regardless of the device's computational power. §5.2 presents the details of scalable DNNs.

4.3 Integrated ABR

Key challenges. As NAS uses per-video DNN, a client must download a DNN from a server to benefit from DNN-based quality enhancement. However, DNN downloads also compete for the bandwidth with the video stream itself. As a result, aggressively downloading the DNN model may degrade user QoE. At the same time, a client may benefit from an early DNN download because it can receive the quality enhancement early on. Because there exists a conflict, a careful decision making as to when and how to download the DNN is critical.

NAS' bitrate adaptation integrates the decision to download a DNN with bitrate selection for QoE maximization. It considers three additional factors that impact user QoE: 1) To benefit from quality enhancement, a client-side DNN must be downloaded first; 2) a partially downloaded DNN improves quality in proportion to the amount downloaded; and 3) DNN chunk downloads compete for bandwidth with video chunk downloads.

To solve the non-trivial problem, we leverage reinforcement learning [52] and generate an ABR algorithm that integrates the decision to download a DNN model. For this, we divide the DNN model into fixed-size chunks and train an RL network that outputs a decision (i.e., whether to download a video or a DNN chunk) using as input the current state (e.g., throughput measurement, playback buffer occupancy, the number of remained video chunks) and its history. We train the RL network using a large training set consisting of real network traces [9, 60].

The decision to use RL brings a number of benefits: 1) it allows NAS to directly optimize for any target QoE metric, while accounting for the benefit of DNN-based quality enhancement; 2) RL balances multiple interacting factors, such as bandwidth variations, bandwidth sharing between video and DNN chunks, and quality enhancement of partial DNNs, in a way that optimizes the QoE; and 3) it naturally accommodates the use of multiple DNNs by encoding the DNN type in the RL network. §5.3 presents the details of the integrated ABR.

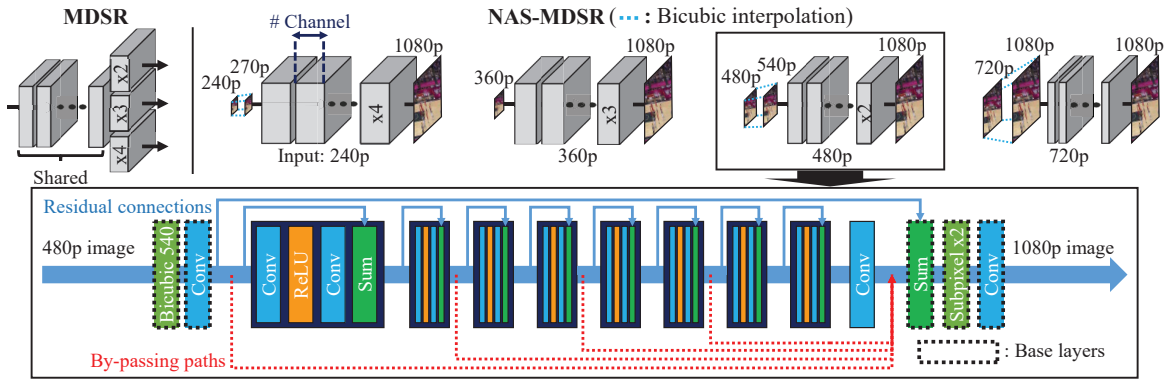


Figure 4: Super-resolution DNN architectures: MDSR vs. NAS-MDSR

5 System Design

NAS is implemented on top of current HTTP adaptive streaming, standardized in DASH [2]. We start by explaining the key differences in how the system operates.

New video admission (server-side processing). As in DASH, when a video clip is uploaded, it is encoded at multiple bitrates and divided into chunks. In addition, the server trains content-aware DNNs for the video for client-side quality enhancement (§5.1). It then associates the DNNs with the video by placing their URLs in the manifest file along with DNN specifications (§5.2).

Client behavior. A client's video player first downloads a manifest file, which contains a list of available DNNs for the video. The client then selects one of them that fits its computing power. The client's DNN processor uses a light-weight mechanism to choose the best available one that fits its resource (§5.2). The player then downloads the selected DNN or video chunks following the decision given by the integrated adaptive bitrate (ABR) algorithm (§5.3). When a DNN chunk is downloaded, the player passes it to the DNN processor, and the processor loads the (partial) DNN on the client's computing device (e.g. GPU). When a video chunk is downloaded, the player keeps it in the playback buffer and the chunk becomes ready for immediate playback. The player then opportunistically passes video chunks in the playback buffer to the DNN processor for quality enhancement along with their associated playback time, which indicates the deadline for neural processing. When the DNN processor finishes processing a chunk, it replaces the original chunk in the playback buffer. Finally, the quality enhanced video chunk is played.

Client-side neural processing. The DNN processor initializes the DNN as DNN chunks arrive. The DNN we use performs quality enhancement on a per-frame basis for super-resolution. Thus, the DNN processor first decodes a video chunk into frames. The DNN processor then applies the super resolution DNN. The resulting frames are then re-encoded to video chunks which replace the original chunks in the playback buffer. The decoding, super-resolution, and encoding phases are pipelined and parallelized to minimize the latency (See §7.5 for details).

5.1 Content-aware DNN for DASH

Applying DNN to adaptive streaming. Standard DNN architectures are not designed for adaptive streaming which introduces specific requirements. First, because adaptive streaming uses multiple resolutions, DNN must be able to take multiple resolutions as input. Second, the DNN inference has to take place in real-time. Finally, DNN should sacrifice its inference quality as little as possible in meeting the first two requirements. The inference time can be reduced at the cost of quality by reducing the number of layers and/or the number of channels (a set of features) in each layer. Such down-scaling also decreases DNN's network footprint. Thus, we must strike a balance between the tradeoff in quality and size, while meeting the real-time requirement.

Using a super-resolution network. Our system extends MDSR [49], a state-of-the-art super-resolution network. As shown in Figure 4, MDSR supports multi-scale super-resolution (x2, x3, x4) in a single network, while sharing the intermediate layers to reduce the footprint. The input resolution drastically affects the inference time of MDSR. For example, even on a flagship desktop GPU (e.g., Nvidia Titan Xp), a 720p input only delivers 3.23 frames per second, whereas a 240p image is processed nearly in real-time at 28.96 frames per second. Thus, to meet the real-time constraint for the highest resolution, one has to downscale the network size.

However, due to the shared layer design, this degrades the quality of all resolutions uniformly, making the lower resolution suffer from significant quality degradation. To avoid the limitation, we use a separate network for each resolution (Figure 4), trading in the total size of the DNN for inference quality. For each DNN, we fix the number

of layers that represents the capacity to adapt to temporal variation in computing power (§5.2). Then, we take the maximum number of channels, independently, for each resolution, that satisfies the real-time constraint. The resulting DNN configuration and size we use for our evaluation are listed in Table 2. The footprint of the 'Ultra-high' DNN is 2,145 KB, which is about the size of a single 1080p (four-second) video chunk (4.8 Mbps) from our evaluation. The size of the 'Low' DNN is only about half the size of a 240p chunk (400 Kbps). While we use NAS-MDSR and specific settings for evaluation, NAS design is not bound to any specific DNNs but accommodates their evolution.

Training content-aware DNNs. The training data is pairs of a low resolution (e.g., 240p, 360p, 480p, 720p) and the original highest resolution (e.g., 1080p) image. We update the DNN parameters to minimize the difference between the DNN's output and the target high resolution image. The training cost is of "one-time" and is amortized across the total watch time of the video episode. Nevertheless, for CDNs that deliver many video episodes, the total computational cost may be high. To reduce the cost of training, we apply the fine-tuning strategy of a transfer learning type. Namely, we first train a generic DNN model using a popular standard benchmark [19]. We then train the content-aware DNN model on each video episode with its weights initialized as those from the generic model. This reduces the computation cost in training by 5-6x, while achieving similar quality enhancement compared to random initialization [33].

5.2 Adaptation to Computational Power

This section describes two mechanisms (multiple DNNs and anytime prediction) that clients use to adapt to their computational capacity to deliver real-time quality enhancement. For each technique, we describe the enabling DNN design and explain the client-side dynamic adaptation logic.

Providing multiple DNNs (server-side). As discussed in §4.2, we provide multiple DNN configurations that vary in quality and computational requirements to support a broad range of GPUs. Similar to GPU's rendering quality options, we provide four quality levels of DNNs: 'Low', 'Medium', 'High', 'Ultra-high'. Thus, we have a DNN per quality per input-resolution, as shown in Table 2. Finally, the server records the available DNN configurations on a manifest file, including the DNN name and level, input resolution, the number of layers, and the number of channels.

Choosing a DNN from multiple options (client-side). Clients test-run the DNN options to choose the one that gives the best quality improvement and delivers real-time performance. A naive way to measure the inference time of DNNs is downloading all DNNs at the client device. However, this consumes large bandwidth (several MBs) and unnecessarily delays video streaming, ultimately degrading user QoE. To streamline the process, NAS provides enough information about the DNN options (i.e., the number of layers and channels) in the manifest file for clients to reconstruct mock DNNs without downloading the DNNs. Using the DNN configuration defined in the manifest file, clients generate DNNs initialized with random weights and run them on their GPUs. Finally, the clients select the largest (highest-quality) DNN that runs in real-time—the client does not need actual weights here because a larger DNN provides better quality. With four DNN options, the client-side test-run takes between 1.64-3.40 seconds depending on a GPU model. Thus, the client can decide which DNN to use early on without downloading any DNN.

Input Resolution	DNN Quality Level			
	Low	Medium	High	Ultra-high
240p	20, 9	20, 21	20, 32	20, 48
	43 KB	203 KB	461 KB	1026 KB
360p	20, 8	20, 18	20, 29	20, 42
	36 KB	157 KB	395 KB	819 KB
480p	20, 4	20, 9	20, 18	20, 26
	12 KB	37 KB	128 KB	259 KB
720p	6, 2	6, 7	6, 16	6, 26
	2 KB	5 KB	17 KB	41 KB

Table 2: DNN configurations for NAS-MDSR (#Layer, #Channel, Size)

Scalable DNN and anytime prediction (server-side). Our scalable DNN architecture enables the client to utilize a partially downloaded DNN and adapt to time-varying computational power. Utilizing partial DNNs provides incremental benefit as the download progresses. This especially benefits the QoE at the beginning of a video because the full DNN of a few MBs cannot be transferred instantly. In addition, the scalable architecture enables anytime prediction allowing us to adapt to client's available computational power that may change unexpectedly.

For this, we modify the DNN architecture and its training method. The DNN's intermediate layers consist of multiple residual blocks [49] each of which consists of two convolutional layers. We allow bypassing consecutive intermediate blocks during inference. To enable bypassing, we add direct connections from intermediate blocks to the final layers, as shown in Figure 4. This creates multiple inference paths as shown in the figure. We then train all interference paths in the following way.

In each training iteration, we randomly bypass intermediate layers to calculate the error between the net-

work output and the target image. Then, we use back-propagation [62] to update parameters. In particular, we go through all layers with probability $1/2$ (for training the original path) and choose one of the remaining by-passing paths uniformly at random otherwise. The resulting DNN can generate an output image using only a part of the DNN and provide incremental quality improvement as more layers are used. Finally, DNNs are divided into chunks. Our server places the chunks' URLs in the video manifest file. The first DNN chunk consists of the base layers for all input resolutions. The subsequent chunks contain the rest of DNNs.

Using the scalable DNN (client-side). A client downloads the DNN in chunks. It reconstructs the first *partial* DNN after downloading the base layers. The size of this minimal DNN is only 35.11% – 36.14% of the full DNN (33 KB to 768 KB), allowing the client to start benefiting from DNN-based quality enhancement early on. As the client downloads more blocks, it updates the DNN, which provides incremental benefit.

Finally, our client opportunistically determines the number of layers to use during video playback. At every time interval, the client's DNN processor first calculates the amount of time remaining until the playback time of the chunk it is processing. The client then calculates the maximum amount of layers it can use that meets the deadline. To aid this, the client records the latest inference time for each layer and updates this table when the inference time changes. We empirically set the time interval to four seconds, which is the length of a single video chunk in our evaluation. This allows NAS clients to dynamically adapt to changes in the available computational power, as we demonstrate in §7.4.

5.3 Integrated Bitrate Adaptation

NAS integrates two decisions into its ABR algorithm for QoE optimization: 1) it decides whether to fetch a video chunk or a DNN chunk; and 2) if the first decision is to fetch a video chunk, it chooses the chunk's bitrate.

The algorithm must balance the two conflicting strategies. The first strategy places emphasis on downloading the DNN model in the hope that this will bring quality enhancement in the future, while sacrificing video's streaming quality at the moment. The second strategy optimizes for video bitrate at the moment and delays the DNN download. In practice, the resulting outcome is unpredictable because it depends on how the network conditions change. The solution space is extremely large considering the number of bitrates, the download order of video and DNN chunks, and the dynamic range of available bandwidth.

To tackle the challenge, we use a reinforcement learning (RL) framework [52,53] that directly optimizes the target metric (without using explicit decision labels) through comprehensive "experience". In particular, we adopt the actor-critic framework of A3C [53]. It learns a strategy (or policy) from observations and produces a mapping from raw observations, such as the fraction of DNN model downloaded, the quality improvement due to DNN, network throughput samples, and playback buffer occupancy, to the decisions described above.

RL design. An RL agent interacts with an environment [68]. For each iteration t, the agent takes an action a_t, after observing a state s_t from the environment. The environment then produces a reward r_t and updates its state to s_{t+1}. A policy is defined as a function that gives the probability of taking action a_t given s_t, $\pi(s_t, a_t) :\rightarrow [0,1]$. The goal then is to learn a policy, π, that maximizes the sum of future discounted reward $\sum_{t=0}^{\infty} \gamma^t r_t$, where $\gamma \in (0,1]$ is a discount-rate for future reward.

In our case, the set of actions $\{a_t\}$ includes whether to download a DNN chunk or to download a video chunk of a specific bitrate. The state s_t includes the number of remaining DNN chunks to download, throughput measurements, and player measurements (e.g., the playback buffer occupancy, past bitrates). Table 3 summarizes the state s_t. The reward r_t is the target QoE metric which is a function of bitrate utility, rebuffering time, and smoothness of selected bitrates [52, 77] defined as:

$$\frac{\sum_{n=1}^{N} q(R_n) - \mu \sum_{n=1}^{N} T_n - \sum_{n=1}^{N-1} \left| q(R_{n+1}) - q(R_n) \right|}{N} \quad (1)$$

where N is the number of video chunks; R_n and T_n respectively represent the video chunk n's bitrate and the rebuffering time resulting from its download; μ is the rebuffering penalty; and $q(R_n)$ is the perceived quality of bitrate R_n (refer to Table 5 in §7.1 for the choices of μ and $q(R_t)$).

To reflect the DNN-based quality enhancement of NAS, we define effective bitrate $R_{\texttt{effective}}$ instead of the nominal bitrate R_n. For each video chunk C_n:

$$R_{\texttt{effective}}(C_n) = \text{SSIM}^{-1}(\text{SSIM}(\text{DNN}_m(C_n)))$$

where $\text{DNN}_m(C_n)$ represents the quality enhanced video chunk C_n after downloading the (partial) DNN chunk m, SSIM is the average structural similarity [73] for measuring the video quality, and its inverse SSIM^{-1} maps a SSIM value back to the video bitrate. To create the mapping, we measure the SSIM of original video chunks at each bitrate (or resolution) and use piece-wise linear interpolation (e.g., (400 Kbps, SSIM_1), ..., (4800 Kbps, SSIM_5)).

Type	State
DNN status	# of remaining DNN chunks
Network status	Throughput for past N chunks
	Download time past N chunks
Player status	Playback buffer occupancy
	Next video chunk sizes
Video status	Bitrate of the latest video chunk
	# of remaining video chunks

Table 3: State used in our RL framework. We use N = 8 which empirically provides good performance.

RL training. Our RL framework has two neural approximators: an actor representing the policy and a critic used to assess the performance of the policy. We use the *policy gradient method* [69] to train the actor and critic networks. The agent first generates trajectories following the current policy $\pi_\theta(s_t, a_t)$, where θ represents parameters (or weights) of the actor's neural network. The critic network observes these trajectories and learns to estimate the *action-value function* $Q^{\pi_\theta}(s_t, a_t)$, which is the total expected reward with respect to taking action a_t starting at state s_t and following the policy π_θ. At each iteration, the actor network uses this estimation to update the model:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t)(Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)),$$

where $V^{\pi_\theta}(s_t)$ is the *value function* representing the total expected reward of π_θ starting at state s_t, and α is the learning rate. In our RL framework, because the reward reflects the average QoE enhancement that content-aware DNN delivers, the critic network learns to estimate the updated total reward. This enables the actor network to learn a policy that balances video and DNN downloads to maximize the QoE.

We use a chunk-level simulator similar to that of Pensieve to accelerate the ABR training. It takes network throughput traces and simulates NAS' video streaming dynamics. In addition, we pre-compute the DNN-based quality enhancement by averaging it over all the videos for each DNN quality. We then use the values to produce a generic ABR model.

When a DNN is downloaded, the simulator updates the amount of downloaded DNN chunks (i.e., decrements the state 'number of remaining DNN chunks'). When a video chunk is downloaded, it adds the chunk to the playback buffer. It then computes the QoE that reflects DNN-based quality enhancement, using the (effective) bitrate utility of each chunk and the rebuffing time. Note, the simulator performs neither actual video downloads nor DNN inferences. Thus, it reduces the training time by 97.12% compared to real-time emulation.

Component	Lines of code (LoC)	Changed
DASH video player	19K lines of JavaScript	8.8% (1763)
Content-aware DNN	6.3K lines of Python	- (6.3K)
Integrated ABR algorithm	5.5K lines of Python	- (5.5K)

Table 4: NAS implementation (Lines of Code)

```
<DNN>
    <Representation quality="low">
      <SegmentTemplate
          DNN="$RepresentationQuality$/$Number$"
                startNumber="1" endNumber="5"/>
    </Representation>
      ...
</DNN>
```

Figure 5: NAS manifest file structure

6 Implementation

We implement NAS client by extending a DASH video player. Both the server-side (training) and client-side (inference) DNN processing are implemented using Pytorch [59]. Table 4 shows the lines of code (LoC) for each component.

NAS client (DASH video player). To implement NAS client, we modify dash.js [4] (version 2.4), a reference implementation of MPEG DASH client written in JavaScript. We run the integrated ABR and content-aware DNNs as separate processes. dash.js is configured to fetch the ABR decisions and quality enhanced chunks through inter-process communication. We add DNN metadata on a manifest file as shown in Figure 5. The `quality` attribute indicates the DNN quality level. The `DNN` attribute of `SegmentTemplate` is used to create the chunk URL, and `startNumber` and `endNumber` indicate the chunk index range. In addition, the manifest file includes the number of layers and the number of channels for each DNN.

Training content-aware DNNs. We implement the scalable super-resolution DNN using Pytorch. For training the DNN model, we use input image patches of size 41x41 pixels by randomly cropping the low-resolution images (e.g., 240p, 360p, 480p, 720p) and run the popular ADAM algorithm [46] to optimize DNN parameters. The mini-batch size, weight decaying parameter, and learning rate are set to 64, 10^{-3}, and 10^{-4}, respectively. We initialize the DNN model using parameters of the generic model (§4.1). We then fine-tune it over 100 mini-batch updates per minute of video to generate a content-aware DNN. Finally, we round off its parameters from single-precision (32-bit) to half-precision (16-bit), which halves the DNN size while introducing minimal performance degradation (virtually no difference in SSIM).

Training integrated ABR. We implement our integrated ABR extending Pensieve's implementation [8]. The initial learning rates of actor/critic networks are set to 10^{-4} and

QoE type	Bitrate utility ($q(R)$)	Rebuffer penalty (μ)
$\text{QoE}_{\texttt{lin}}$	R	4.3
$\text{QoE}_{\texttt{log}}$	$log(R/R_{min})$	2.66
$\text{QoE}_{\texttt{hd}}$	0.4→1, 0.8→2, 1.2→3 2.4→12, 4.8→15	8

Table 5: QoE metrics used for evaluation

10^{-3}, respectively. The entropy weight is initialized as 2. We iterate training over 60,000 epochs in which we decay the entropy weight from 2 to 0.055 exponentially over every epoch. Finally, we select the ABR network that delivers the highest QoE for our training traces.

7 Evaluation

We evaluate NAS by answering the following questions.

- How does NAS perform compared to its baseline competitors, and what is the training cost?
- How does each design component of NAS contribute to the overall performance?
- Does NAS effectively adapt to heterogeneous devices and temporal variance in client's computing power?
- What is the end-to-end processing latency and resource usage of NAS?

7.1 Methodology

Videos. We use videos from popular channels on Youtube. For each of the nine Youtube channel categories, we select three popular channels in the order of appearance. We then pick the most popular video from each channel that supports 1080p quality and whose length is longer than 5 minutes—the number of views of 27 video clips ranges from 7M to 737M. Finally, we download 1080p videos and produce multi-bitrate videos following the Youtube and DASH recommendations [3,13]: Each 1080p video is re-encoded using the H.264 codec [11] in which GOP (or chunk size), frame rate, and bitrates are respectively set to 4 seconds, 24 fps and {400, 800, 1200, 2400, 4800}Kbps (which represent for {240, 360, 480, 720, 1080}p resolution videos). Unless otherwise noted, we use the entire video for training and use the first 5 minutes for playback. Training over the entire video ensures that NAS delivers consistent quality enhancement over the entire video.

Network traces. We use a real bandwidth dataset consisting of 508 throughput traces from Norway's 3G network (2010-2011) [60] and 421 traces from U.S. broadband (2016) [9], compiled by the Pensieve author [8]. We filter out the traces that consistently experience low bandwidth ($<$ 400 Kbps) for an extended time ($\geq$100 seconds). The resulting average throughput ranges from 0.38 Mbps to 4.69 Mbp, and the mean and median are 1.31 Mbps and 1.09 Mbps, respectively. Each trace spans 320 seconds, and we loop the trace until a video is completely downloaded. We use randomly selected 80 % of our traces for training and the remaining 20 % for testing.

Baseline. We compare NAS against the following state-of-the-art bitrate adaptation that does not utilize client computation.

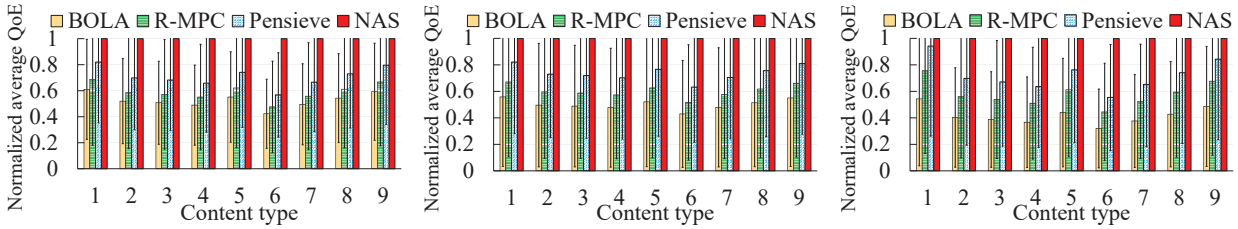
- Pensieve [52] uses deep reinforcement learning to maximize QoE.
- RobustMPC [77] uses playback buffer occupancy and throughput predictions over next five chunks to select the bitrate that maximizes QoE. We use the version reproduced by the authors of Pensieve [8].
- BOLA [66] uses Lyapunov optimization based on playback buffer occupancy. We use the BOLA version implemented in dash.js, which is a Javascript-based reference implementation of a MPEG-DASH player [4].

QoE metrics. We use three types QoE metrics, compiled by MPC and Pensieve, whose the bitrate utility function, $q(R_n)$, and rebuffering penalty constant, μ of Equation 1, differ as summarized in Table 5.

- $\text{QoE}_{\texttt{lin}}$ uses a linear bitrate utility.
- $\text{QoE}_{\texttt{log}}$ uses a logarithmic bitrate utility function that represents its decreasing marginal utility.
- $\text{QoE}_{\texttt{hd}}$ heavily favors high-definition (HD) video (720p and 1080p) over non-HD.

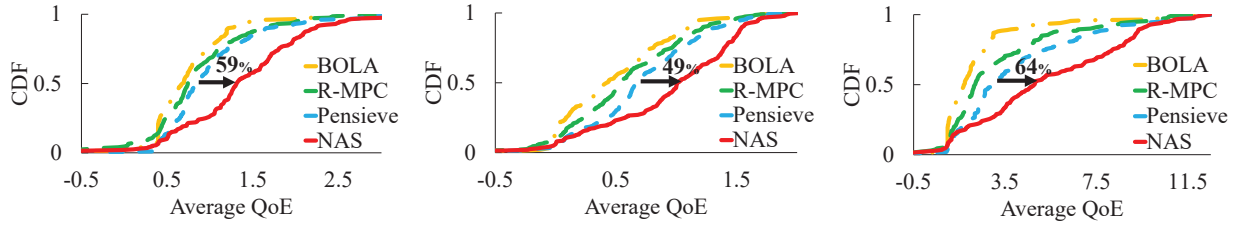
Experimental settings. We run our dash.js implementation on a Chromium Browser (version 65) to stream MPEG-DASH videos. We use six GPU models from Nvidia's desktop GPU product line listed in Table 1. Unless otherwise noted, the default client-side GPU is Nvidia Titan Xp. In our setting, the content-aware DNN and the ABR network run at the client as separate processes. To emulate the network conditions from the network traces, we use Mahimahi [55].

We use two experiment settings. To evaluate NAS client on all six GPUs, we have a local testbed. To scale training and testing, we use Google Cloud Platform. Training is done using GPU instances equipped with Nvidia's server-class Tesla P100 GPU. However, Google Cloud Platform does not have desktop class GPUs, while we need to scale client-side streaming experiments to 18 hours of network traces x 27 video clips x 4 types of ABR x 3 types of QoE, totaling 5,832 hours of streaming time. Thus, we take quality and latency measurements of content-aware DNNs using the local testbed on each GPU device for each video. We then emulate the network condition between NAS server and client once for each network trace and apply the effect of the quality enhancement and latency of content-aware DNNs. We confirm the network-emulated, DNN-simulated clients produce



(a) QoE_{lin} (b) QoE_{log} (c) QoE_{hd}

Figure 6: Normalized QoE comparison of video clips from the nine content categories of YouTube. (1: Beauty, 2: Comedy, 3: Cook, 4: Entertainment, 5: Game, 6: Music, 7: News, 8: Sports, 9: Technology)



(a) QoE_{lin} (b) QoE_{log} (c) QoE_{hd}

Figure 7: Cumulative distribution of QoE for 'Sports' content category

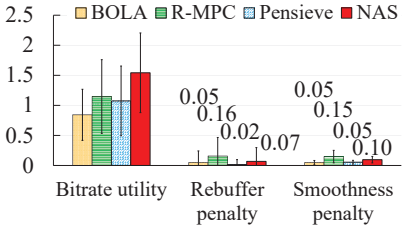


Figure 8: QoE_{lin} breakdown

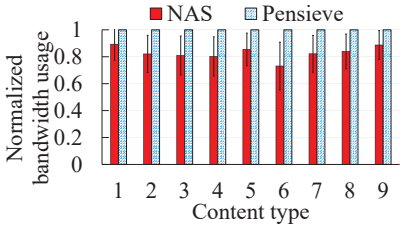


Figure 9: Normalized bandwidth usage at QoE_{lin}=0.98

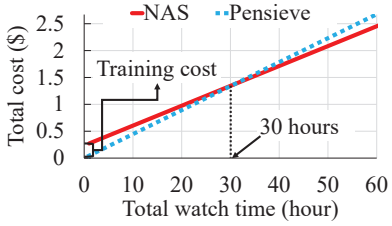


Figure 10: Cumulative server cost

the same QoE as real clients of our local testbed using a fraction of test data.

7.2 NAS vs. Existing Video Delivery

QoE improvement. Figure 6 shows the average QoE of video clips across the nine content categories. The error bars indicate one standard deviation from the average. NAS delivers the highest QoE across all content categories over all three QoE metrics. The result shows significant improvement over prior work. NAS consistently outperforms Pensieve by a large margin across all QoE metrics: QoE_{lin} (43.08% better), QoE_{log} (36.26% better), and QoE_{hd} (42.57% better). With QoE_{lin}, NAS outperforms Pensieve 43.08% on average, whereas Pensieve achieves a 19.31% improvement over RobustMPC (R-MPC). Compared to BOLA, NAS achieves 92.28% improvement in QoE_{lin}. The QoE improvement varies across content types from 21.89% (1: 'Beauty') to 76.04% (6: 'Music') over Pensieve because many factors, such as the scene complexity, compression artifacts, and temporal redundancy, affect the DNN performance.

Figure 7 shows the cumulative distribution of QoE over our test traces. It shows the 'Sports' content category which shows medium gain among all categories. NAS delivers benefit across all network conditions. NAS improves the median QoE_{lin} by 58.55% over Pensieve. Note, Pensieve mainly delivers its QoE gain over RobustMPC by reducing rebuffering at the cost of bitrate utility. In contrast, NAS does not exhibit such tradeoff because it uses client computation. Other content (not shown) displays a similar trend. Finally, Figure 8 shows a breakdown of QoE into bitrate utility, rebuffering penalty, and the smoothness penalty. NAS benefits the most from the bitrate utility due to the DNN-based quality enhancement.

Bandwidth savings. Despite the DNN transfer overhead, NAS requires less bandwidth in delivering the same QoE level. To demonstrate this, we create a hypothetical setting using the chunk-level simulator (§5.3) where NAS clients receive a fraction of bandwidth that Pensieve clients receive including the DNN transfer overhead. We adjust the fraction and empirically determine the fraction that deliv-

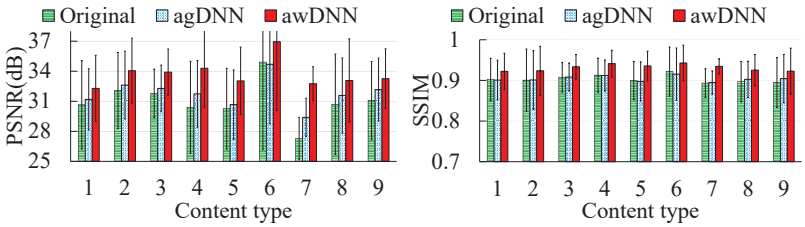


Figure 11: Video quality in PSNR and SSIM (240p input)

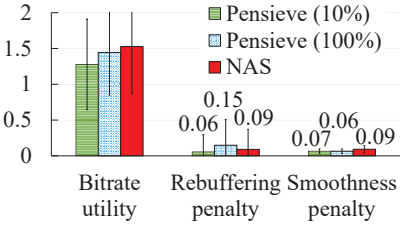
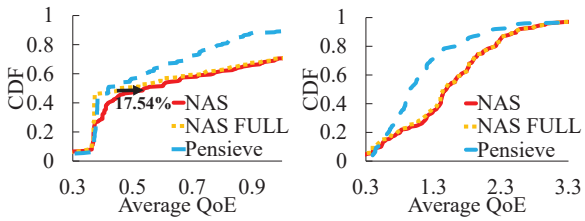


Figure 12: Integrated ABR vs. Baseline algorithm



(a) $t \leq 47.82$ seconds (b) $t > 47.82$ seconds

Figure 13: Scalable DNN vs. Full DNN

ers the same QoE. We assume NAS clients download the largest DNN ('Ultra-high') model for every five-minute video. Figure 9 shows the average bandwidth usage of Pensieve and NAS. On average across all videos, NAS requires 17.13% less bandwidth than Pensieve to deliver the same quality. The savings vary across content between 10.69% and 26.90%. This demonstrates the benefit of using a DNN outweighs the overhead of transferring the DNN.

Cost-benefit analysis (server-side). We now quantify the overall server-side cost in using a NAS content delivery network. While NAS servers use less bandwidth to deliver the same quality, they must train content-aware DNNs and the integrated ABR network. We quantify the computation and bandwidth cost of the CDN servers. The training time for the integrated ABR is only 10.92 hours on a CPU. Because it is a one-time cost amortized across all video streams, the additional cost is negligible. In contrast, the content-aware DNNs must be trained for each video. The total training time (across multiple DNNs) per minute of video is 10 minutes.

For a Google cloud instance with 8 vCPUs, 32 GB RAM, and a Nvidia P100 GPU, this translates to $0.23 per minute of video. For bandwidth, Amazon CDN instance charges at most 0.085 $/GB. The price per bandwidth becomes cheaper as one uses more bandwidth. Using these as reference, we compute the total video delivery cost a function of cumulative viewing time per minute of video. Figure 10 shows the cost comparison for NAS and Pensieve. As before, we assume each user watches a video clip for five minutes (i.e., DNNs are transferred every five minutes of viewing). This is a conservative estimate given the popularity of binge-watching [5]. NAS pays the upfront cost of computation, but as the cumulative viewing time increases, it is amortized. Note, NAS uses 17.13% less bandwidth to deliver the same user QoE. Thus, when the cumulative viewing reaches 30 hours (per minute of video in the system), NAS CDN recoups the initial investment.

7.3 Component-wise Analysis

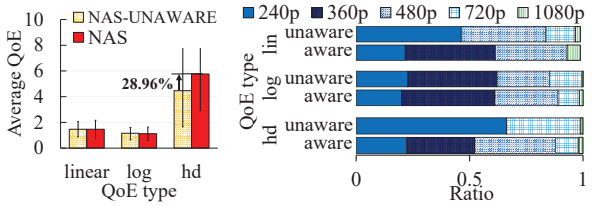
We evaluate how each design component contributes to the quality improvement.

Content-awareness. Figure 11 compares video quality of content-aware DNN (awDNN), a content-agnostic DNN (agDNN) trained on standard benchmark images (NTIRE 2017 dataset [19]), and the original 240p video we use as input upscaled by the bicubic interpolation. We measure the video quality both in PSNR [35] and SSIM [73] in which PSNR represents the average mean square error between two images in logarithmic decibel scale. Content-aware DNN delivers consistent improvement whereas content-agnostic DNNs even degrades the quality in some cases with respect to the PNSR measure (content type: 6) and the SSIM measure (type: 1,2,4,5,6). This confirms our rationale for using DNN's training accuracy.

Scalable DNN. Figure 13 demonstrates the benefit of utilizing a partial DNN. We compare Pensieve, NAS, and a version of NAS (`NAS-FULL`) that does not utilize partial DNN downloads. Specifically, Figure 13(a) shows the cumulative distribution of QoE_{lin} before the average full DNN download time (47.82 seconds). As soon as a partial DNN is downloaded (22.16 seconds on average), NAS enhances the quality. The result shows that this delivers 17.54 % and 3.35 % QoE improvement in the median and mean, respectively. Note, the QoE of NAS and `NAS-FULL` becomes identical after downloading the full DNN ($t > 47.82$ seconds) as shown in Figure 13(b).

Integrated ABR. The integrated ABR delivers benefit during and after the DNN download. To demonstrate this, we create two hypothetical settings using the chunk-level simulator (§5.3).

First, we compare NAS with a version that uses a naive



(a) Average QoE (b) Video quality distribution

Figure 14: Integrated ABR vs. Quality-unaware ABR

Model Name	Frames per second (FPS)			
	Low	Medium	High	Ultra-high
GTX 1050 Ti	**34.36**	17.62	14.91	7.37
GTX 1060	45.27	**30.05**	25.96	13.17
GTX 1070 Ti	41.76	45.24	**41.53**	21.47
GTX 1080	53.82	52.86	**38.95**	21.46
GTX 1080 Ti	58.94	56.29	57.12	**31.34**
Titan Xp	52.99	51.72	52.22	**33.58**

Table 6: DNN processing speed on desktop GPUs (Bold font indicates the selected quality level.)

DNN download strategy that downloads a DNN at a fraction of the video bitrate chosen by Pensieve. Note, it does not integrate the bitrate selection and DNN download decision. We use two variants: one that aggressively downloads DNN at 100% of the video bitrate and the other that uses only 10%. Both are configured to start downloading the DNN when the playback buffer becomes larger than 15.42 seconds, which is the average time that NAS starts to stream a DNN in our test traffic traces. Our result shows NAS respectively outperforms the non-aggressive and aggressive strawman by 16.36% and 9.13% with respect to QoE_{lin}. Figure 12 shows the comparison of QoE components. The non-aggressive version experiences lower bitrate utility compared to NAS because the former downloads the DNN more slowly. In contrast, the aggressive version increases the rebuffering penalty by x2.5 which negatively affects the QoE.

Next, to evaluate the benefit of quality-enhancement aware bitrate selection after the DNN is fully downloaded, we compare NAS with a quality-enhancement unaware ABR after the full DNN download. Figure 14(a) shows the average QoE in this setting. We see that the quality-enhancement aware ABR delivers a large gain for QoE_{hd} (28.96 %), whereas it offers minimal benefit to QoE_{lin} (0.01 %) and slightly degrades on QoE_{log} (-3.14 %). The reason it delivers a large gain for QoE_{hd} is because the DNN-enhanced quality of 1.2 Mbps (480p) videos get close to that of the original 2.4 Mbps (720p) video and the marginal utility with respect to the increased quality is far greater for QoE_{hd} than any other QoE type, especially between 1.2 Mbps to 2.4 Mbps (Table 5). The integrated

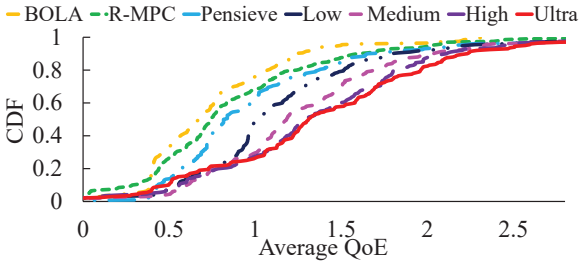


Figure 15: CDF of QoE over different quality DNNs

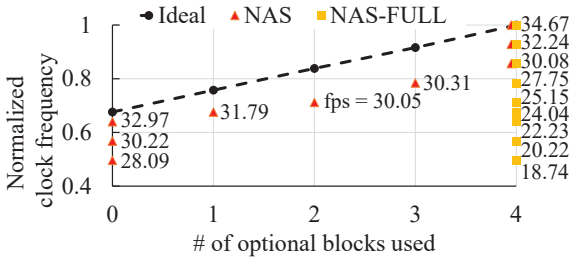


Figure 16: Dynamic adaptation to computing power

ABR reflects this enhancement in the bitrate decisions to download 480p much more often when using QoE_{hd} as shown in Figure 14(b).

7.4 Dynamic Adaptation to Computation

We demonstrate NAS's ability to adapt to heterogeneous clients and temporal variation in computing power.

Heterogeneous clients. We demonstrate NAS is able to meet real-time constraints on six desktop GPUs shown in Table 1. We run our DASH client on six clients each with a different GPU model and measure their performance. First, we measure the throughput of the DNN processing engine, which includes decoding, DNN inference, and re-encoding. Table 6 reports the minimum processing speed across all input resolutions for each device. The video playback rate is 30 frames per second. Clients perform a test-run when it receives a video manifest file. The selected quality level (e.g., 'Low', 'Medium', 'High', or 'Ultra-high') is indicated in boldface. We see that each device selects one that meets the real-time constraint. Note the processing time does not depend on video content.

Next, we measure the QoE of clients using four different quality levels in our cloud setting. Figure 15 shows the cumulative distribution of QoE_{lin} for each quality level. All quality levels outperform Pensieve. The higher the quality level DNN, the better the quality it delivers. Note, even though DNNs of higher quality are larger in size, they deliver incremental benefit over lower quality DNNs.

In sum, the results indicate that NAS adapts to heterogeneous devices, and a device with higher computational power receives greater benefit.

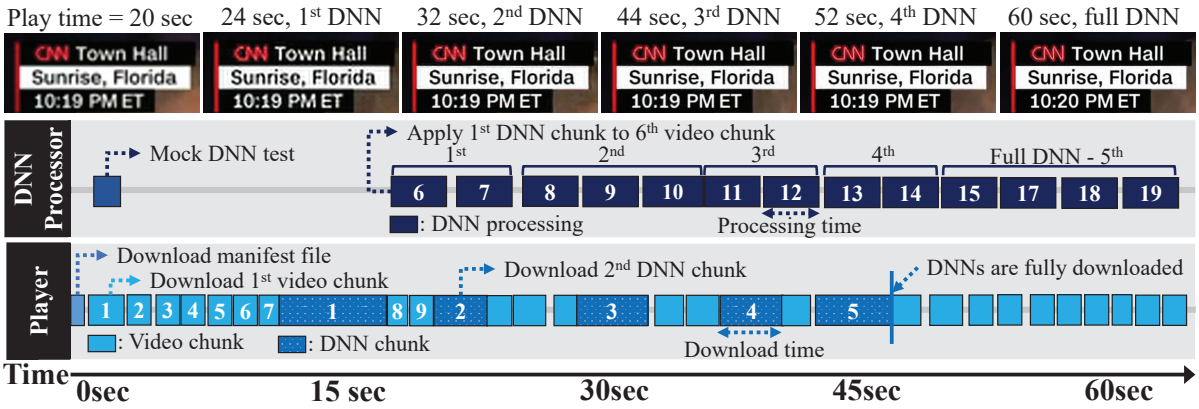


Figure 17: Case study: A time-line of NAS client in operation (Video Source: [16])

Phase	Processing time (sec)
Decode	0.28
Super resolution	3.69
Encode	0.91
Total	4.88
NAS' parallel pipeline	3.76 (23.0% reduction)

Table 7: Video processing time per phase

Temporal variation. We evaluate client's adaptation to temporal variation in computing power in isolation. We emulate the changes in available computing power by varying the clock frequency of GPU (Titan Xp). We change the GPU clock frequency at 10% granularity and report the inference path used and its throughput. Figure 16 shows the result compared to a naive version that only uses the full DNN (`NAS-FULL`) and the ideal line that plots the normalized throughput (y-axis) of each inference path at full clock cycle. x-axis shows the number of optional blocks used for inference. The y-intercept represents the computing requirement for the required layers of DNN. We report the raw throughput for all results. It shows NAS adapts to the changing resource with anytime prediction to the extent that the underlying DNN supports and thus delivers real-time performance unlike `NAS-FULL` that does not.

7.5 End-to-end Operation

NAS client in operation. We present an end-to-end operation of our NAS client using a network trace from our testset. We use a client with a Titan Xp GPU, running on our local testbed. Figure 17 shows the time-line starting from a video request made from our DASH client. At $t = 0.36$ (sec), it downloads the video manifest and test-runs the mock DNNs to select the DNN quality level. The test-run finishes at $t = 2$. The video starts playing at $t = 2.03$. At $t = 17.64$, the first DNN chunk is downloaded, and the minimal DNN initialized at $t = 17.71$. At this time, the DNN processing begins, and video chunks (6-7) in the playback buffer receive quality enhancement. Subsequent video chunks are processed by the DNN as they arrive. As new DNN chunks arrive, the DNNs are incrementally updated. At $t = 46.41$, DNNs are fully downloaded.

DNN processing time. We evaluate the DNN processing latency of NAS client. For DNN processing, NAS pipelines three processes, each of which respectively handles decoding, super-resolution, and re-encoding. Table 7 shows the processing time for each phase for a four-second video chunk. We use GTX 1080 Ti for processing 'Ultra-high' quality DNN using a 30 fps, 240p video as input. We re-encode the DNN's output in H.264 using the fastest option in ffmpeg [6]. This is because the compression factor is not important. The total processing time when each phase is serialized is 4.88 seconds, whereas our pipelined processing takes 3.76 seconds. Considering super-resolution takes 3.69 seconds, the latency overhead of the rest is minimal.

Finally, we measure the client's GPU memory usage for DNN processing. 'Ultra-high', 'High', 'Medium', 'Low' quality DNNs respectively use 3.57 GB, 3.12 GB, 3.05 GB, and 2.99 GB of GPU memory.

8 Conclusion

We present NAS, a video delivery system that utilizes client computation to enhance the video quality. Unlike existing video delivery that solely relies on the bandwidth resource, NAS uses client-side computation powered by deep neural networks (DNNs). NAS introduces new system designs to address practical problems in realizing the vision on top of DASH. Our evaluation over real videos on real network traces shows NAS delivers improvement between 21.89–76.04% in user quality of experience (QoE) over the current state of the art. Finally, the cost-benefit analysis shows content distribution networks can actually reduce the cost of video delivery while providing the same or better QoE compared to the current state of the art.

9 Acknowledgments

We thank our shepherd Wyatt Lloyd and anonymous reviewers for their constructive feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (Ministry of Science and ICT) [No.2018-0-00693] and [No.R-20160222-002755]. Dongsu Han is the corresponding author.

References

[1] Apple HTTP live streaming official homepage. https://developer.apple.com/streaming/.

[2] Dash industry forum. https://dashif.org/.

[3] Dash recommend segment length. https://bitmovin.com/mpeg-dash-hls-segment-length/.

[4] dash.js Github repository. https://github.com/Dash-Industry-Forum/dash.js.

[5] Deloitte: 73 Percent of Americans Binge Watch TV; Millennial Binge Watchers Average Six Episodes and Five Hours per Viewing. https://www.prnewswire.com/news-releases/deloitte-73-percent-of-americans-binge-watch-tv-millennial-binge-watchers-average-six-episodes-and-five-hours-per-viewing-300427152.html.

[6] FFmpeg - H.264 Video Encoding Guide. https://trac.ffmpeg.org/wiki/Encode/H.264.

[7] The iPhone X's new neural engine exemplifies Apple's approach to AI. https://www.theverge.com/2017/9/13/16300464/apple-iphone-x-ai-neural-engine.

[8] Pensieve official Github repository. https://github.com/hongzimao/pensieve.

[9] Raw Data - Measuring Broadband America 2016. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016.

[10] The state of online video 2017. https://www.limelight.com/resources/white-paper/state-of-online-video-2017/#weeklyconsumption.

[11] Video codec - H.264 standardization. https://www.itu.int/rec/T-REC-H.264/.

[12] Video codec - H.265 standardization. http://x265.org/.

[13] Youtube recommended upload encoding settings. https://support.google.com/youtube/answer/1722171?hl=en.

[14] Everything Wrong With Frozen In 10 Minutes Or Less. https://www.youtube.com/watch?v=HvwMtWkfkJ8, June 2014.

[15] Minecraft Xbox - School Day [244]. https://www.youtube.com/watch?v=5N6E2cF-CGw, Nov. 2014.

[16] Shooting survivor confronts NRA spokesperson Dana Loesch. https://www.youtube.com/watch?v=4AtOU0dDXv8, Feb. 2018.

[17] ADHIKARI, V. K., GUO, Y., HAO, F., HILT, V., ZHANG, Z. L., VARVELLO, M., AND STEINER, M. Measurement study of netflix, hulu, and a tale of three cdns. *IEEE/ACM Transactions on Networking (ToN) 23*, 6 (Dec 2015), 1984–1997.

[18] AGUSTSSON, E., MENTZER, F., TSCHANNEN, M., CAVIGELLI, L., TIMOFTE, R., BENINI, L., AND GOOL, L. V. Soft-to-hard vector quantization for end-to-end learning compressible representations. In *Advances in Neural Information Processing Systems* (2017), pp. 1141–1151.

[19] AGUSTSSON, E., AND TIMOFTE, R. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (July 2017).

[20] AKHSHABI, S., BEGEN, A. C., AND DOVROLIS, C. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http. In *Proceedings of the ACM Multimedia Systems Conference (MMSys)* (2011), ACM, pp. 157–168.

[21] AKHTAR, Z., NAM, Y. S., RAO, S., AND RIBEIRO, B. Oboe : Auto-tuning video abr algorithms to network conditions. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2018).

[22] BALACHANDRAN, A., SEKAR, V., AKELLA, A., SESHAN, S., STOICA, I., AND ZHANG, H. Developing a predictive model of quality of experience for internet video. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2013).

[23] BENGIO, Y. Learning deep architectures for AI. *Foundations and Trends in Machine Learning 2*, 1 (2009), 1–127.

[24] BOLUKBASI, T., WANG, J., DEKEL, O., AND SALIGRAMA, V. Adaptive neural networks for efficient inference. In *Proceedings of the International Conference on Machine Learning (ICML)* (2017), pp. 527–536.

[25] CHANG, H., YEUNG, D.-Y., AND XIONG, Y. Super-resolution through neighbor embedding. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on* (2004), vol. 1, IEEE, pp. I–I.

[26] CISCO. Cisco visual networking index: Forecast and methodology, 20162021. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf, July 2017.

[27] DOBRIAN, F., SEKAR, V., AWAN, A., STOICA, I., JOSEPH, D., GANJAM, A., ZHAN, J., AND ZHANG, H. Understanding the impact of video quality on user engagement. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2011).

[28] DONG, C., LOY, C. C., HE, K., AND TANG, X. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence 38*, 2 (2016), 295–307.

[29] FATAHALIAN, K. The rise of mobile visual computing systems. *IEEE Pervasive Computing 15*, 2 (Apr 2016), 8–13.

[30] FITZEK, F. H. P., AND REISSLEIN, M. Mpeg-4 and h.263 video traces for network performance evaluation. *IEEE Network 15*, 6 (Nov 2001), 40–54.

[31] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2017), USENIX Association, pp. 363–376.

[32] GANJAM, A., SIDDIQUI, F., ZHAN, J., LIU, X., STOICA, I., JIANG, J., SEKAR, V., AND ZHANG, H. C3: Internet-scale control plane for video quality optimization. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2015), vol. 15, pp. 131–144.

[33] HE, K., ZHANG, X., REN, S., AND SUN, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (2015), pp. 1026–1034.

[34] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[35] HORE, A., AND ZIOU, D. Image quality metrics: Psnr vs. ssim. In *Pattern recognition (icpr), 2010 20th international conference on* (2010), IEEE, pp. 2366–2369.

[36] HU, H., DEY, D., HEBERT, M., AND BAGNELL, J. A. Anytime neural network: a versatile trade-off between computation and accuracy. *arXiv preprint arXiv:1708.06832* (2018).

[37] HUANG, Q., ANG, P., KNOWLES, P., NYKIEL, T., TVERDOKHLIB, I., YAJURVEDI, A., DAPOLITO, IV, P., YAN, X., BYKOV, M., LIANG, C., TALWAR, M., MATHUR, A., KULKARNI, S., BURKE, M., AND LLOYD, W. Sve: Distributed video processing at facebook scale. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2017), ACM, pp. 87–103.

[38] HUANG, S., PAPERNOT, N., GOODFELLOW, I., DUAN, Y., AND ABBEEL, P. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284* (2017).

[39] HUANG, T.-Y., HANDIGOL, N., HELLER, B., MCKEOWN, N., AND JOHARI, R. Confused, timid, and unstable: Picking a video streaming rate is hard. In *Proceedings of the Internet Measurement Conference (IMC)* (2012), pp. 225–238.

[40] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[41] JIANG, J., SEKAR, V., MILNER, H., SHEPHERD, D., STOICA, I., AND ZHANG, H. Cfa: A practical prediction system for video qoe optimization. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2016), pp. 137–150.

[42] JIANG, J., SEKAR, V., AND ZHANG, H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).

[43] JIANG, J., SEKAR, V., AND ZHANG, H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN) 22*, 1 (2014), 326–340.

[44] JIANG, J., SUN, S., SEKAR, V., AND ZHANG, H. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2017), vol. 1, p. 3.

[45] KIM, J., KWON LEE, J., AND MU LEE, K. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 1646–1654.

[46] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[47] KRISHNAN, S. S., AND SITARAMAN, R. K. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking (ToN) 21*, 6 (2013), 2001–2014.

[48] LEE, H., AND SHIN, J. Anytime neural prediction via slicing networks vertically, 2018.

[49] LIM, B., SON, S., KIM, H., NAH, S., AND LEE, K. M. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (2017).

[50] LIU, H. H., WANG, Y., YANG, Y. R., WANG, H., AND TIAN, C. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2012).

[51] LIU, X., DOBRIAN, F., MILNER, H., JIANG, J., SEKAR, V., STOICA, I., AND ZHANG, H. A case for a coordinated internet video control plane. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2012), pp. 359–370.

[52] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2017), pp. 197–210.

[53] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)* (2016), pp. 1928–1937.

[54] MUKERJEE, M. K., NAYLOR, D., JIANG, J., HAN, D., SESHAN, S., AND ZHANG, H. Practical, real-time centralized control for cdn-based live video delivery. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015), pp. 311–324.

[55] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for http. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2015), pp. 417–429.

[56] NIKLAUS, S., MAI, L., AND LIU, F. Video frame interpolation via adaptive convolution. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

[57] OOYALA. Ooyala global video index q1 2017. `http://go.ooyala.com/rs/447-EQK-225/images/Ooyala-Global-Video-Index-Q1-2017.pdf`. Last accessed: July 2017.

[58] PAPERNOT, N., MCDANIEL, P., JHA, S., FREDRIKSON, M., CELIK, Z. B., AND SWAMI, A. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE.

[59] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS) Workshop* (2017).

[60] RIISER, H., VIGMOSTAD, P., GRIWODZ, C., AND HALVORSEN, P. Commute path bandwidth traces from 3g networks: analysis and applications. In *Proceedings of the ACM Multimedia Systems Conference (MMSys)* (2013), pp. 114–118.

[61] RIPPEL, O., AND BOURDEV, L. Real-time adaptive image compression. In *International Conference on Machine Learning* (2017).

[62] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533.

[63] SANDVINE. 2016 global internet phenomena report: North america and latin america. https://www.sandvine.com/downloads/general/global-internet-phenomena/2016/global-internet-phenomena-report-latin-america-and-north-america.pdf. Last accessed: July 2017.

[64] SCHULTER, S., LEISTNER, C., AND BISCHOF, H. Fast and accurate image upscaling with super-resolution forests. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 3791–3799.

[65] SHI, W., CABALLERO, J., LEDIG, C., ZHUANG, X., BAI, W., BHATIA, K., DE MARVAO, A. M. S. M., DAWES, T., OREGAN, D., AND RUECKERT, D. Cardiac image super-resolution with global correspondence using multi-atlas patchmatch. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (2013), Springer, pp. 9–16.

[66] SPITERI, K., URGAONKAR, R., AND SITARAMAN, R. K. Bola: Near-optimal bitrate adaptation for online videos. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)* (2016), IEEE, pp. 1–9.

[67] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[68] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[69] SUTTON, R. S., MCALLESTER, D. A., SINGH, S. P., AND MANSOUR, Y. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)* (2000), pp. 1057–1063.

[70] TANG, L., HUANG, Q., PUNTAMBEKAR, A., VIGFUSSON, Y., LLOYD, W., AND LI, K. Popularity prediction of facebook videos for higher quality streaming. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[71] TODERICI, G., VINCENT, D., JOHNSTON, N., HWANG, S. J., MINNEN, D., SHOR, J., AND COVELL, M. Full resolution image compression with recurrent neural networks. In *International Conference on Machine Learning* (2017).

[72] TSCHANNEN, M., AGUSTSSON, E., AND LUCIC, M. Deep generative models for distribution-preserving lossy compression. *arXiv preprint arXiv:1805.11057* (2018).

[73] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[74] WENDELL, P., JIANG, J. W., FREEDMAN, M. J., AND REXFORD, J. Donar: Decentralized server selection for cloud services. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2010).

[75] WOLPERT, D. H. The lack of a priori distinctions between learning algorithms. *Neural Computation 8*, 7 (1996), 1341–1390.

[76] YEO, H., DO, S., AND HAN, D. How will deep learning change internet video delivery? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 57–64.

[77] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015).

[78] ZOU, W. W., AND YUEN, P. C. Very low resolution face recognition problem. *IEEE Transactions on Image Processing 21*, 1 (2012), 327–340.

Floem: A Programming System for NIC-Accelerated Network Applications

Phitchaya Mangpo Phothilimthana
University of California, Berkeley

Ming Liu
University of Washington

Antoine Kaufmann
University of Washington

Simon Peter
The University of Texas at Austin

Rastislav Bodik
University of Washington

Thomas Anderson
University of Washington

Abstract

Developing server applications that offload computation to a NIC accelerator is complex and laborious. Developers have to explore the design space, which includes semantic changes for different offloading strategies, as well as variations on parallelization, program-to-resource mapping, and communication strategies for program components across devices.

We therefore design FLOEM — a language, compiler, and runtime — for programming NIC-accelerated applications. FLOEM enables offload design exploration by providing programming abstractions to assign computation to hardware resources; control mapping of logical queues to physical queues; access fields of a packet and its metadata without manually marshaling a packet; use a NIC to memoize expensive computation; and interface with an external application. The compiler infers which data must be transferred between the CPU and NIC and generates a complete cache implementation, while the runtime transparently optimizes DMA throughput. We use FLOEM to explore NIC-offloading designs of real-world applications, including a key-value store and a distributed real-time data analytics system; improve their throughput by 1.3–3.6× and by 75–96%, respectively, over a CPU-only implementation.

1 Introduction

Network bandwidth is growing much faster than CPU performance [5], forcing many data-center applications to sacrifice application cycles for packet processing [9, 23, 37]. As a result, system developers have started to offload computation to programmable network interface controllers (NICs), dramatically improving the performance and energy efficiency of many data-center applications, such as search engines, key-value stores, real-time data analytics, and intrusion detection [12, 23, 26, 40]. These NICs have a variety of hardware architectures including FPGAs [12, 33, 48], specialized flow engines [6], and more general-purpose network processors [3, 32].

However, implementing data-center network applications in a combined CPU-NIC environment is difficult. It often requires many design-implement-test iterations before the accelerated application can outperform its CPU-only version. These iterations involve non-trivial changes: programmers may have to move portions of application code across the CPU-NIC boundary and manually refactor the program.

We propose FLOEM, a programming system for NIC-accelerated applications. Our current prototype targets a platform with the Cavium LiquidIO [3], a general-purpose programmable NIC that executes C code. FLOEM is based on a data-flow language that is natural for expressing packet processing logic and mapping *elements* (modular program components) onto hardware devices. The language lets developers easily move an element onto a CPU or a NIC to explore alternative offloading designs, as well as parallelize program components. Application developers can define a FLOEM element as a Python class that contains a C implementation of the element. To aid programming productivity, we provide a library of common elements.

Further examining how developers offload data-center applications to NICs, we have identified the following commonly encountered problems, which led us to propose abstractions and mechanisms amenable to a data-flow programming model that can solve these problems.

- Different offloading choices require different communication strategies. We observe that these strategies can be expressed by a **mapping of logical communication queues to physical queues**, so we propose this mapping as a part of our language.

- Moving computation across the CPU-NIC boundary may change which parts of a packet must be sent across the boundary. Marshaling the necessary packet fields

is tedious and error-prone. Thus, we propose **per-packet state** — an abstraction that allows a packet and its metadata to be accessed anywhere in the program — while FLOEM automatically transfers only required packet parts between a NIC and CPU.

- Using an in-network processor to cache application state or computation is a common pattern for accelerating data-center applications. However, it is non-trivial to implement a cache that guarantees the consistency of data between a CPU and NIC. We propose a **caching construct** for memoizing a program region, relieving programmers from having to implement a complete cache protocol.
- Developers often want to **offload an existing application** without rewriting the code into a new language. We let programmers embed C code in elements and allow a legacy application to interact with FLOEM elements via a simple function call, executing those elements in the host process of the legacy application.

We demonstrate that without significant programming effort, FLOEM can help offload parts of real-world applications — a key-value store and a real-time analytics system — improving their throughput by 1.3–3.6× and 75–96%, respectively, over a CPU-only configuration.

In summary, this paper makes the following contributions:

- Identifying *challenges* in designing of NIC-accelerated data-center applications (Section 2)
- Introducing *programming abstractions* to address these challenges (Sections 3 and 4)
- Developing a programming system that enables exploration of alternative offloading designs, including a *compiler* (Section 5) and a *runtime* (Section 6) for efficient data transfer between a CPU and NIC

2 Design Goals and Rationale

We design FLOEM to help programmers explore how to offload their server network applications to a NIC. The applications that benefit from FLOEM have computations that *may* be more efficient to run on the NIC than on the CPU because of the NIC's hardware-accelerated functions, parallelism, or reduced latency when eliminating the CPU from fast-path processing. These computations include packet filtering (e.g., format validation and classification), packet transformation (e.g., serialization, compression, and encryption), packet steering (e.g., load balancing to CPU cores), packet generation, and caching of application state. This list is not exhaustive. Ultimately, we would like FLOEM to help developers discover new ways to accelerate their applications.

The main challenge when designing programming abstractions is to realize a small number of constructs that let programmers express a large variety of implementation choices. This requires an understanding of common challenges within the application domain. We build FLOEM to meet the following design goals.

Goal 1: Expressing Packet Processing

As described above, computations suitable for NIC offloading are largely packet processing. Programming abstractions and systems for packet processing have long been studied, and the Click modular router [34] is widely used for this task. We adopt its data-flow model to ease the development of packet processing logic (Section 3).

Goal 2: Exploring Offload Designs

A data-flow model is suitable for mapping computations to desired hardware devices, as we have seen with many Click extensions that support offloading [24, 27, 46]. Similarly, FLOEM programmers implement functionality once, as a data-flow program, after which they can use code annotations to assign elements to desired devices and to parallelize the program. However, trivially adopting a data-flow model is insufficient to meet this design goal. By inspecting the design of a key-value store and a TCP stack offloaded with FlexNIC [23], we discover several challenges that shape the design of our language.

Logical-to-physical queue mapping (Section 4.1). One major part of designing an offloading strategy is managing the transfer of data between the host and accelerator. Various offloading strategies require different communication strategies, such as how to steer packets, how to share communication resources among different types of messages, and whether to impose an order of messages over a communication channel.

By examining hand-optimized offloads, we find that developers typically express communication in terms of logical queues and then manually implement them using the provided hardware communication mechanisms. A logical queue handles messages sent from one element to another, while a hardware communication channel implements one physical queue. As part of an offload implementation, developers have to make various mapping choices among logical and physical queues. The right mapping depends on the workload and hardware configuration and is typically realized via trial-and-error.

To aid this task, we design a queue construct with an explicit logical-to-physical queue mapping that can be controlled via parameters and by changing element connections. Existing frameworks [24, 27, 46] do not support this mapping. To control the number of physical

queues in these frameworks, programmers have to explicitly: (1) create more logical queues by demultiplexing the flow into multiple branches and making more elements and connections, or (2) merge logical queues by multiplexing multiple branches into one.

Per-packet state (Section 4.2). In a well-optimized program, developers meticulously construct a message by copying only the necessary parts of a packet to send between a CPU and NIC; this minimizes the amount of data transferred over PCIe. When developers move computation between the CPU and NIC, they may need to rethink which fields must be sent, slowing the exploration of alternative offloading designs.

Nevertheless, no existing system performs this optimization automatically. ClickNP [27] sends an entire packet, while NBA [24] and Snap [46] rely on developers to annotate each element with a packet's *region of interest*, specified as numeric offsets in a packet buffer. We design FLOEM to automatically infer what data to send across the CPU-NIC boundary and offer the *per-packet state* abstraction as if an entire packet could be accessed anywhere in the program. This abstraction resembles P4's per-packet metadata [10] and RPC IDLs (e.g., XDR [14] and Google's protobuf [18]). However, P4 allows per-packet metadata to be carried across multiple processing pipelines only within a single device, while RPC IDLs generate marshaling code based on interface descriptions, rather than automatically inferring.

Caching construct (Section 4.3). Caching application state or memoizing computation in an in-network processor is a common strategy to accelerate server applications [15, 22, 26, 30]. While the abstractions we have so far are sufficient to express this strategy, implementing a cache protocol still requires a significant effort to guarantee both data consistency and high performance when messages between a CPU and NIC may arrive out-of-order. Thus, we introduce a *caching construct*, a general abstraction for caching that integrates well with the data-flow model. This construct provides a full cache protocol that maintains data consistency between the CPU and NIC. Unlike FLOEM, existing systems support caching only of flow state [6, 27] — which typically does not require maintaining consistency between the CPU and NIC — but not caching of application state.

Goal 3: Integrating with Existing Applications

Prior frameworks were designed exclusively to implement network functions and packet processing [13, 16, 24, 27, 34, 36, 46], where computation is mostly stateless and simpler than in our target domain of server applications. While parts of typical server applications can be built by composing pre-defined elements, many parts cannot. In our target domain, developers often want to offload an application by reusing existing application code instead of writing code from scratch. Besides porting existing applications, some developers may prefer to implement most of their applications in C because a data-flow programming model may not be ideal for the full implementation of complex applications.

FLOEM lets developers combine custom and stock elements, embed C code in data-flow elements, and integrate a FLOEM program with an external program. As a result, developers can port only program parts that may benefit from offloading into the data-flow model. The impedance mismatch between the data-flow model and the external program's model (e.g., event-driven or imperative) raises the issue of interoperability. Our solution builds on the queue construct to decouple the internal part from the interface part, which appears to the external program as a function (**Section 4.4**). The external program can execute the function using its own thread to (1) retrieve a message from the queue and process it through elements in the interface part, or (2) process a message through the interface part and push it to the queue.

3 Core Abstractions

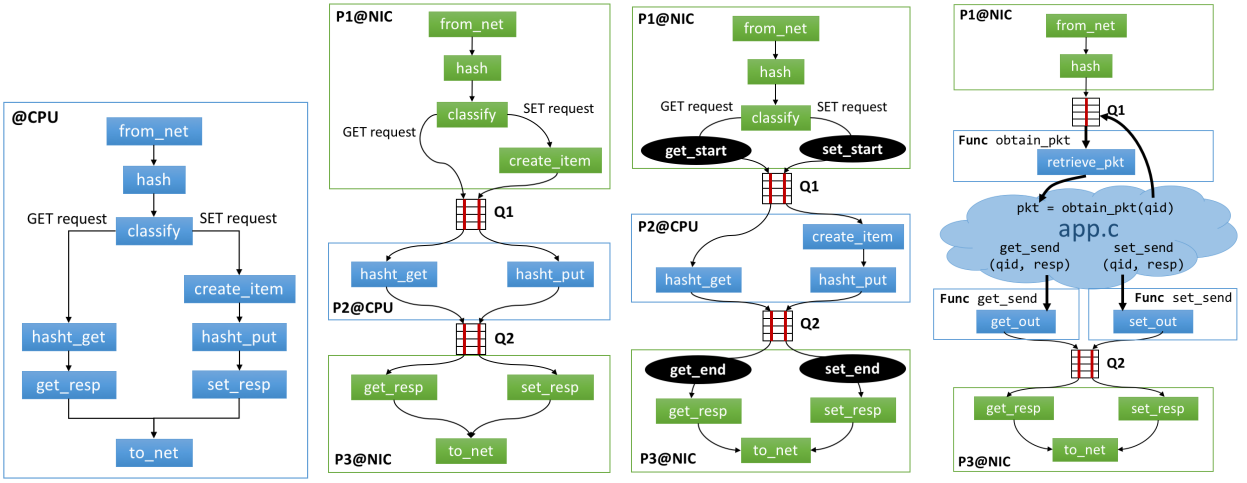
We use a key-value store application as our running example. Figure 1 displays several offloading designs for the applicaton: CPU-only (Figure 1a), split CPU-NIC (Figure 1b), and NIC as cache (Figure 1c). Figure 1d illustrates how to create an interface that an external program can use to interact with FLOEM. We show how to implement these offloads using our programming abstractions in this and the next sections.

Elements. FLOEM programs are composed of elements. Upon receiving inputs from all its input ports, an element processes the inputs and emits outputs to its output ports. The listing below illustrates how to create the `classify` element in our key-value store example, which classifies incoming requests by type (GET or SET).

```
class Classify(Element): # Define an element class
  def configure(self):
    self.inp = Input(pointer(kvs_message))
    self.get = Output(pointer(kvs_message))
    self.set = Output(pointer(kvs_message))

  def impl(self):
    self.run_c(r'''   // C code
      kvs_message *p = inp();
      uint8_t cmd = p->mcr.request.opcode;

      output switch { // switch --> emit one output port
        case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
        case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
      }
    ''')
classify = Classify()  # Instantiate an element
```



(a) No offloading

(b) CPU-NIC split (see Section 4.1)

(c) NIC as cache (see Section 4.3)

(d) Interface to external program (see Section 4.4)

Figure 1: Several offloading strategies of a key-value store implemented in FLOEM

We specify input and output ports in the `configure` method. We express the logic for processing a single packet in the `impl` method by calling `run_c`, which accepts C code with special syntax to retrieve value(s) from an input port and emit value(s) to an output port.

To create the program shown in Figure 1a, we connect elements as follows:

```
from_net >> hash >> classify
classify.get >> hasht_get >> get_resp >> to_net
classify.set >> item >> hasht_put >> set_resp >> to_net
```

Note that `.get` and `.set` refer to the output ports of `classify`.

Queues. Instead of pushing data to the next element instantaneously, a queue can store data until the next element dequeues it. A queue can connect and send data between elements on both different devices (e.g., CPU and NIC) and on the same device.

Shared state. FLOEM provides a shared state abstraction that lets multiple elements share a set of variables that are persistent across packets. For example, elements `hasht_get` and `hasht_put` share the same state containing a hash table. FLOEM normally prohibits elements on different devices from sharing the same state. Instead, programmers must use message passing across queues to share information between those elements. Shared state lets programmers express complex stateful applications.

Segmented execution model. A *segment* is a set of connected elements that begins with from a *source* element, which is either a `from_net` element or a queue, and ends with *leaf* elements (elements with no output ports) or queues. A queue sends packets between segments. Our execution model is run-to-completion within a segment. A source element processes a packet and pushes it to subsequent elements until the packet reaches the end of the segment. When the entire segment finishes processing a packet, it starts on the next one. By default, one thread on a CPU executes each segment, so elements within a segment run sequentially with respect to their data-flow dependencies.

The program in Figure 1a has a single segment, while the program in Figure 1b has three. Note that not all elements in a segment must be executed for each packet. In our example, either `hasht_get` or `hasht_put` (not both) will be executed depending on the port where `classify` pushes a packet to.

Offloading and parallelizing. A segment is a unit of code migration and parallelization. Programmers map each segment to a specific device by supplying the `device` parameter. They can also assign multiple threads to run the same segment to process different packets in parallel using the `cores` parameter. Programmers cannot assign a segment to run on both the NIC and CPU in parallel; the current workaround is to create two identical segments, one for NIC and another for CPU. Figure 2 displays a FLOEM program that implements a sharded key-value store with the offloading strategy in Figure 1b.

4 Advanced Offload Abstractions

This section presents programming abstractions that we propose to mitigate recurring programming challenges encounters when exploring different ways to offload applications to a NIC.

```
1  Q1 = Queue(channel=2, inst=3)
2  Q2 = Queue(channel=2, inst=3)
3
4  class P1(Segment):
5    def impl(self):
6      from_net >> hash >> queue_id >> classify
7      classify.get >> Q1.enq[0]  # channel 0
8      classify.set >> create_item >> Q1.enq[1] # chnl 1
9
10 class P2(Segment):
11   def impl(self):
12     self.core_id >> Q1.qid  # use core id as queue id
13     Q1.deq[0] >> hasht_get >> Q2.enq[0]
14     Q1.deq[1] >> hasht_put >> Q2.enq[1]
15
16 class P3(Segment):
17   def impl(self):
18     scheduler >> Q2.qid # scheduler produces queue id
19     Q2.deq[0] >> get_resp >> to_net
20     Q2.deq[1] >> set_resp >> to_net
21
22 P1(device=NIC, cores=[0,1]) # run on core id 0,1
23 P2(device=CPU, cores=[0,1,2])
24 P3(device=NIC, cores=[2,3])
```

Figure 2: FLOEM program implementing a sharded key-value store with the CPU-NIC split strategy of Figure 1b

4.1 Logical-to-Physical Queue Mapping

To achieve correctness and maximize performance, FLOEM gives programmers control over how the compiler instantiates logical queues for a particular offloading strategy. The queue construct `Queue(channel=n, inst=m)` represents n logical queues (n channels) using m physical queues (m instances). For example, `Q1` on line 1 of Figure 2 represents two logical queues — displayed as red channels in Figure 1b — using three physical queues. Different mappings of logical to physical queues lead to different communication strategies, as elaborated below.

Packet steering. Developers can easily implement packet steering by creating a queue with multiple physical instances. For example, in the split CPU-NIC version of the key-value store (Figure 1b), we want to shard the key-value store so that different CPU threads can handle different subsets of keys to avoid lock contention and CPU cache misses. As a result, we want to represent queue `Q1` by multiple physical queues, with each CPU thread having a dedicated physical queue to handle requests for its shard. The NIC then steers a packet to the correct physical queue based on its key. FlexNIC [23] shows that such key-based steering improves throughput of the key-value store application by 30–45%.

To implement this strategy, we create `Q1` with multiple physical queues (line 1 in Figure 2). Steering a packet is controlled by assigning the target queue instance ID to the `qid` field of *per-packet state* in the C code of any element that precedes the queue. In this example, we set `state.qid = hash(pkt.key) % 3`, where `state` refers to *per-packet state*.

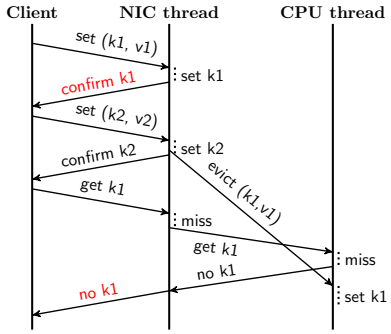


Figure 3: Inconsistency of a write-back cache if messages from NIC to CPU are reordered

Resource sharing. Developers may want to map multiple logical queues to the same physical queue for resource sharing, or vice versa for resource isolation. For example, they may want to consolidate infrequently used logical queues into one physical queue to obtain a larger batch of messages per PCIe transfer. In the sharded key-value store, we want to use the same physical queue to transport both the GET and SET requests of one shard so that the receiver's side processes these requests at the same rate as the sender's. To implement this, we use `Q1` to represent two logical queues (line 1 in Figure 2): one for GET and one for SET. Different degrees of sharing can vary application performance by up to 16% (Section 7.2).

Packet ordering. For correctness, developers may want to preserve the order of packets being processed from one device to another. For example, an alternative way to offload the key-value store is to use the NIC as a key-value cache, only forwarding misses to the CPU. To ensure consistency of the write-back cache, we must enforce that the CPU handles evictions and misses of the same key in the same order as the cache. Figure 3 shows an inconsistent outcome when an eviction and a miss are reordered. To avoid this problem, developers can map logical queues for evictions and misses to the same physical queue, ensuring in-order delivery.

The ability to freely map logical to physical queues lets programmers express different communication strategies with minimal effort in a declarative fashion. A queue can also be parameterized by whether its enqueuing process is lossless or lossy, where a lossless queue is blocking. Note that programmers are responsible for correctly handling multiple blocking queues.

4.2 Per-Packet State

FLOEM provides per-packet state, an abstraction that allows access to a packet and its metadata from any element without explicitly passing the state. To use this

abstraction, programmers define its format and refer to it using the keyword `state`. For our key-value store, we define the format of the per-packet state as follows:

```
class MyState(State): # define fields in a state
 hash = Field(uint32_t)
 pkt  = Field(pointer(kvs_message))
 key  = Field(pointer(void), size='state.pkt->keylen')
```

The provided element `from_net` creates a per-packet state and stores a packet pointer to `state.pkt` so that subsequent elements can access the packet fields, such as `state.pkt->keylen`. The element `hash` computes the hash value of a packet's key and stores it in `state.hash`, which is used later by element `hasht_get`. To handle a variable-size field, FLOEM requires programmers to specify its size, as with the key field above.

4.3 Caching Construct

With only minimal changes to a program, FLOEM offers developers a high-level caching construct for exploring caching on the NIC and storing outputs of expensive computation to be used in the future. First, programmers instantiate the caching construct `Cache` to create an instance of a cache storage and elements `get_start`, `get_end`, `set_start`, and `set_end`. Programmers then insert `get_start` right before the get query begins, and `get_end` right after the get query ends; a get query is computation we want to memoize. Programmers must also specify what to store as a key (input) and a value (output) in the cache; this can be done by assigning `state.key` and `state.keylen` (key and keylen fields of per-packet state) before the element `get_start`, and assigning `state.val` and `state.vallen` before `get_end`. If the application has a corresponding set query, elements `set_start` and `set_end` must be inserted, and those fields of the per-packet state must be assigned accordingly for the set query; a set query mutates application state and must be executed when a cache eviction occurs. Finally, programmers can use parameters to configure the cache with the desired table size, cache policy (either write-through or write-back), and a write-miss policy (either write-allocate or no-write-allocate).

For our key-value store example, we can use the NIC to cache outputs from hash table get operations by just inserting the caching elements, as shown in Figure 1c. Notice that queues `Q1` and `Q2` are parts of the expensive queries (between `get_start` and `get_end` and between `set_start` and `set_end`) that can be avoided if outputs are in the cache.

Requirements. The get and set query regions cannot contain any *callable segment* (see Section 4.4). Elements `get_start`, `get_end`, `set_start`, and `set_end` must be on the same device. Paths between `get_start` and `get_end`, and between `set_start` and `set_end`, must pass through the same set of queues (e.g., Figure 1c) to ensure the in-order delivery of misses and evictions of the same key. Multiple caches can be used as long as cached regions are not overlapped. The compiler returns an error if a program violates these requirements.

4.4 Interfacing with External Code

To help developers offload parts of existing programs to run on a NIC, we let them: (1) embed C code in elements, (2) implement elements that call external C functions available in linkable object files, and (3) expose segments of FLOEM elements as functions callable from any C program. The first mechanism is the standard way to implement an element. The second simply links FLOEM-generated C code with object files. For the last mechanism, we introduce a *callable segment*, which contains elements between a queue and an endpoint, or vice versa. An endpoint element may send/receive a value to/from an external program through its output/input port. A callable segment is exposed as a function that can be called by an external program to execute the elements in a segment.

In Figure 1d, we implement simple computation, such as hashing and response packet construction, in FLOEM, but we leave complex functionality, including the hash table and item allocation, in an external C program. The external program interacts with the FLOEM program to retrieve a packet, send a get response, and send a set response via function `obtain_pkt`, `get_send`, and `set_send`, respectively. The following listing defines the function `obtain_pkt` using a callable segment. This function takes a physical queue ID as input, pulls the next entry from the queue with the given ID, executes element `retrieve_pkt` on the entry, and returns the output from `retrieve_pkt` as the function's return value.

```
class ObtainPkt(CallableSegment):
  def configure(self):
    self.inp = Input(int)      # argument is int
    self.out = Output(q_entry) # return value is q_entry

  def impl(self):
    self.inp >> Q1.qid
    Q1.deq >> retrieve_pkt >> self.out

ObtainPkt(name='obtain_pkt')
```

The external program running on the CPU calls `obtain_pkt` to retrieve a packet that has been processed by element `hash` on the NIC and pushed into queue `Q1`.

5 The FLOEM Compiler

The FLOEM compiler contains three primary components that: (1) translate a data-flow program with elements into C programs, (2) infer minimal data transfers across queues, and (3) expand the high-level caching construct into primitive elements, as depicted in Figure 4.

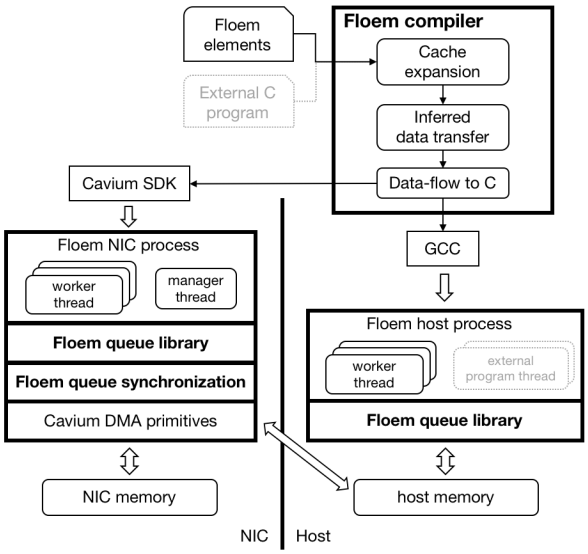
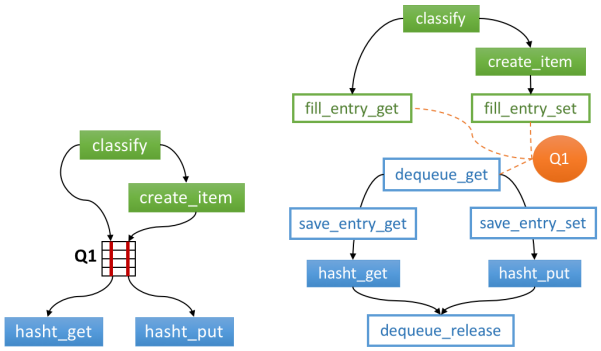


Figure 4: FLOEM system architecture



(a) Before transformation (b) After transformation

Figure 5: The key-value store's data-flow subgraph in the proximity of queue `Q1` from the split CPU-NIC version

5.1 Data-Flow to C

FLOEM compiles a data-flow program into two executable C programs: one running on the CPU and the other on the NIC. Our code generator compiles a segment of primitive elements into a chain of function calls, where one element corresponds to a function. The compiler replaces an output port invocation with a function call to the next element connected to that output port. The calling element passes an output value to the next element as an argument to the function call. Earlier compiler passes transform queues (Section 5.2) and caching constructs (Section 5.3) into primitive elements.

5.2 Inferred Data Transfer

In this section, we explain how the FLOEM compiler infers which fields of a packet and its metadata must be sent across each queue, and how it transforms queues into a set of primitive elements.

Liveness analysis. The compiler infers per-packet state's fields to send across each logical queue (each queue's channel) using a classical liveness analysis [7]. The analysis collects used and defined fields at each element and propagates information backward to compute a *live* set at each element (i.e., a set of fields that are used by the element's successors). For each segment, the compiler also collects a *use* set of all fields that are accessed in the segment.

Transformation. After completing the liveness analysis, the compiler transforms each queue construct into multiple primitive elements that implement enqueue and dequeue operations. In the split CPU-NIC version of the key-value store example, the compiler transforms queue `Q1` in Figure 5a into the elements in Figure 5b.

To enqueue an entry to a logical queue at a channel `X`, we first create element `fill_entry_X` to reserve a space in a physical queue specified by `state.qid`. We then copy the *live* per-packet state's fields at channel `X` into the queue. To dequeue an entry, element `dequeue_get` locates the next entry in a specified physical queue, classifies which channel the entry belongs to, and passes the entry to the corresponding output port (i.e., demultiplexing). Element `save_entry_X` allocates memory for the per-packet state on the receiver's side to store the *use* fields and a pointer to the queue entry so that the fields in the entry can be accessed later. Each `save_entry_X` is connected to the element that was originally connected to that particular queue channel. Finally, the compiler inserts a `dequeue_release` element to release the queue entry after its last use in the segment. These generated elements utilize the built-in queue implementations described in Section 6.

5.3 Cache Expansion

The compiler expands each high-level caching construct into primitive elements that implement a cache policy using the expansion rules shown in Figure 6. Each node in the figure corresponds to a subgraph of one or more elements. For a write-through cache without allocation on write misses, the compiler expands the program graphs that handle get and set queries in the left column into the graphs in the middle column. For a write-back policy with allocation on write misses, the resulting graphs are shown in the right column. For get-only applications, we skip the set expansion rule.

We apply various optimizations to reduce response time. For example, when a new allocation causes an

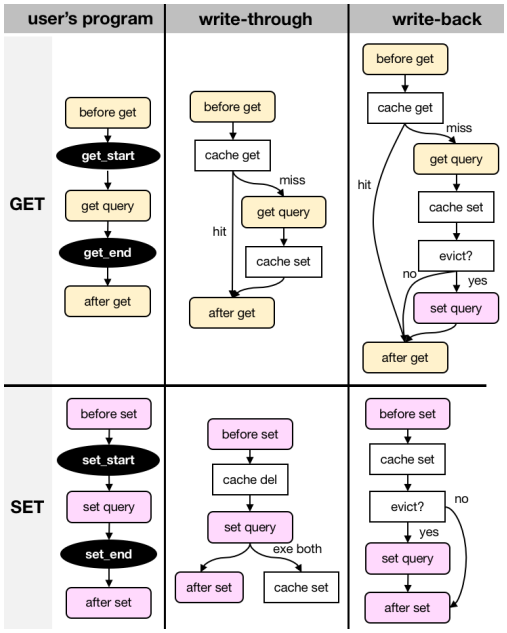


Figure 6: Cache expansion rules

eviction in a write-back cache, we write back the evicted key asynchronously. Instead of waiting for the entire `set query` to finish before executing `after get` (e.g., sending the response), we wait only until the local part of `set query` (on a NIC) reaches a queue to the remote part of `set query` (on a CPU). Once we successfully enqueue the eviction, we immediately execute `after get`.

5.4 Supported Targets

We prototype FLOEM on a platform with a Cavium LiquidIO NIC [3]. We use GCC and Cavium SDK [2] to compile C programs generated by FLOEM to run on a CPU in user mode and on a NIC, respectively. If a FLOEM program contains an interface to an external C program, the compiler generates a C object file that the external application can link to in order to call the interface functions.

Intrinsics, libraries, and system APIs of the two hardware targets differ. To handle these differences, FLOEM lets programmers supply different implementations of a single element class to target x86 and Cavium via `impl` and `impl_cavium` methods, respectively. If `impl_cavium` is not implemented, the compiler refers to `impl` to generate code for both targets. To generate programs with parallelism, FLOEM uses *pthread* on the CPU for multiple segments and relies on the OS thread scheduler. On the NIC, we directly use hardware threads and assign each segment to a dedicated NIC core. Consequently, the compiler prohibits creating more segments on the NIC than the maximum number of cores (12 for LiquidIO).

6 PCIe I/O Communication

To efficiently communicate between the NIC and CPU over PCIe, FLOEM provides high-performance, built-in queue implementations, which rely on the queue synchronization layer (sync layer) to efficiently synchronize data between NIC and CPU. Figure 4 depicts how these components interact with the rest of the system. Currently, we support only a one-way queue with fixed-size entries, parameterized during compile-time.

6.1 Queue Synchronization Layer

Because DMA engines on the NIC are underpowered, they must be managed carefully. If we implemented the queue logic together with data synchronization, the queue implementation would be extremely complicated and difficult to troubleshoot. Hence, we decouple these layers. The sync layer can then additionally be used for other queue implementations, such as a queue with variable-size entries.

Our sync layer provides the illusion that the NIC writes directly to a circular buffer in host memory, where one buffer represents one physical queue. The layer keeps shadow copies of queues in local NIC memory, asynchronously synchronizes these copies with master copies in host memory, batches multiple DMA requests, and overlaps DMA operations with other computation.

To use this layer, a queue implementation must: (1) maintain a status flag in each entry to indicate its availability, and (2) provide basic queue information and queue entry's status checking functions. In turn, the sync layer provides `access_entry` and `access_done` functions to the queue implementation; the queue implementation must call `access_entry` and `access_done` before and after accessing/modifying any queue entry, respectively.

6.2 Maintaining Coherent Buffers

The queue synchronization layer relies on FLOEM's NIC runtime to maintain coherence between buffers on the NIC and the CPU by taking advantage of the circular access pattern of reads followed by writes. We do not explicitly track a queue's head and tail; instead, we use a status flag in each entry to determine if an entry is filled or empty. We choose this design to synchronize both the queue entry's content and status using one DMA operation instead of two. Thus, the runtime continuously checks the state of every queue entry and performs actions accordingly.

Typically, a queue entry on the NIC cycles through *invalid*, *reading*, *valid*, *modified*, and *writing* states, as shown in Figure 7. An *invalid* entry contains stale content and must be fetched from host memory. An asyn-

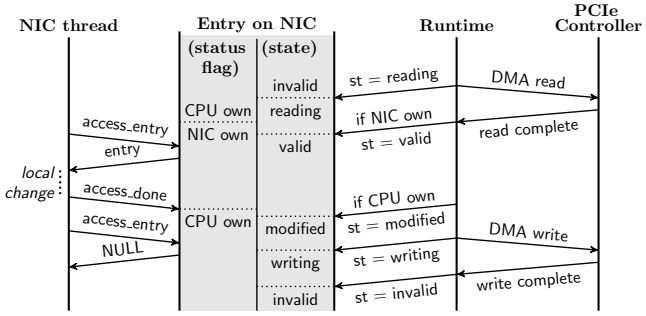


Figure 7: Transitions of a queue entry's status by a NIC worker thread and a NIC runtime manager thread

chronous DMA read transitions an entry from *invalid* to *reading* state. Once the read completes, and the entry is NIC owned (indicated by the status flag), the entry transitions to *valid* state. It may transition back to *invalid* if it is still CPU owned, for example, when the NIC attempts to dequeue an entry that the CPU has not finished enqueuing. The runtime uses the status checking functions provided by the queue implementation to check an entry's status flag. The program running on the NIC can access only *valid* entries; function `access_entry` returns the pointer to an entry if it is in *valid* state; otherwise, it returns NULL.

An entry transitions from *valid* to *modified* once the queue implementation calls function `access_done` to indicate that it is finished accessing that entry. An asynchronous DMA write then transitions the entry to *invalid* state, based on the assumption that the CPU side will eventually modify it, and the NIC must read it from the CPU. This completes a typical cycle of states through which an entry passes.

Note that the CPU side does not need this sync layer or track these states because, unlike the NIC, it does not issue DMA operations.

6.3 I/O Batching

In the actual implementation, we do not track the state of individual queue entries due to high overhead. Instead, we use five pointers to divide a circular queue buffer into five portions with the five states. When a pointer advances, we effectively change the states of a batch of entries that the pointer has moved past. The runtime has a dedicated routine to advance each pointer, and executes these routines in round-robin fashion, overlapping DMA read/write routines with other routines. To achieve DMA batching, the DMA read routine issues a DMA read for the next batch of entries instead of a single entry, as does the DMA write routine. We use a configurable number of dedicated NIC cores (manager threads) to execute the runtime. Each core manages a disjoint subset of queues.

More details about our queue implementation and queue synchronization layer beyond this section can be found in Section 3.6 of the first author's thesis [38].

7 Evaluation

We ran experiments on two small-scale clusters to evaluate the benefit of offloading on servers with different generations of CPUs: 6-core Intel X5650 in our *Westmere* cluster, and 12-core Intel E5-2680 v3 in our *Sandy Bridge* cluster (more powerful). Each cluster had four servers; two were equipped with Cavium LiquidIO NICs, and the others had Intel X710 NICs. All NICs had two 10Gbps ports.

We evaluated CPU-only implementations on the servers with the Intel X710 NICs, using DPDK [4] to send and receive packets bypassing the OS networking stack to minimize overheads. We used the servers with the Cavium LiquidIO NICs to evaluate implementations with NIC offloading. The Cavium LiquidIO has a 12-core 1.20GHz cnMIPS64 processor, a set of on-chip/off-chip accelerators (e.g., encryption/decryption engines), and 4GB of on-board memory.

7.1 Programming Abstraction

We implemented in FLOEM two complex applications (key-value store and real-time data analytics) and three less complex network functions (encryption, flow classification, and network sequencer).

Hypothesis 1 FLOEM *lets programmers easily explore offload strategies to improve application performance.*

The main purpose of this experiment is to demonstrate that FLOEM makes it easier to explore alternative offloading designs, *not* to show when or how one should or should not offload an application to a NIC.

For the complex applications, we started with a CPU-only solution as a baseline by porting parts of an existing C implementation into FLOEM. Then, we used FLOEM to obtain a simple partition of the application between the CPU and NIC for the first offload design. In both case studies, we found that the first offloading attempt was unsuccessful because an application's actual performance can greatly differ from a conceptual estimate. However, we used FLOEM to redesign the offload strategy to obtain a more intelligent and higher performing solution, with minimal code changes, and achieved 1.3–3.6× higher throughput than the CPU-only version.

For the less complex workloads, FLOEM let us quickly determine whether we should dedicate a CPU core to handle the workload or just use the NIC and save CPU cycles for other applications. By merely changing FLOEM's device mapping parameter, we found that

it was reasonable to offload encryption and flow classification to the NIC, but that the network sequencer should be run on the CPU. The rest of this section describes the applications in our experiment in greater detail.

Case Study: Key-Value Store

In this case study, we used one server to run the key-value store and another to run a client generating workload, communicating via UDP. The workload consisted of 100,000 key-value pairs of 32-byte keys and 64-byte values, with the Zipf distribution ($s = 0.9$) of 90% GET requests and 10% SET requests, the same workload used in FlexNIC [23]. We used a single CPU core with a NIC offload (potentially with multiple NIC cores); this setup was reasonable since other CPU cores may be used to execute other applications simultaneously. Figure 8 shows the measured throughput of different offloading strategies, and Table 1 summarizes the implementation effort.

CPU-only (Figure 1a): We ported an existing C implementation, which runs on a CPU using DPDK, into FLOEM except for the garbage collector of freed key-value items. This effort involved converting the original control-flow logic into the data-flow logic, replacing 538 lines of code with 334 lines. The code reduction came from using reusable elements (e.g., `from_net` and `to_net`), so we did not have to set up DPDK manually.

***Split CPU-NIC (Figure 1b)*:** We tried a simple CPU-NIC partition, following the offloading design of FlexKVS [23], by modifying 296 lines of the CPU-only version; this offload strategy was carefully designed to minimize computational cycles on a CPU. It required many changes because the NIC (`create_item` element) creates key-value items that reside in CPU memory. Unexpectedly, this offload strategy lowered performance (the second bar). Profiling the application revealed a major bottleneck in the element that prepares a GET response on the NIC. The element issued a blocking DMA read to retrieve the item's content from host memory. This DMA read was not part of queue `Q2` because that queue sent only the pointer to the item, not the item itself. Therefore, the runtime could not manage this DMA read; as a result, this strategy suffered from this additional DMA cost.

***NIC caching (Figure 1c)*:** We then used FLOEM to explore a completely different offload design. Since the Cavium NIC has a large amount of local memory, we could cache a signification portion of the key-value store on the NIC. This offload design, previously explored, was shown to have high performance [26]. Therefore, we modified the CPU-only version by inserting the caching construct (43 lines of code) as well as creating segments and inserting queues (62 lines of code). For a baseline comparison, code relevant to communication on the CPU side alone was already at 240 lines in a manually-written C implementation of FlexKVS with a software NIC emulation. This translated to fewer than 15 lines of code in FLOEM. These numbers show that implementing a NIC-offload application without FLOEM requires significantly more effort than with FLOEM.

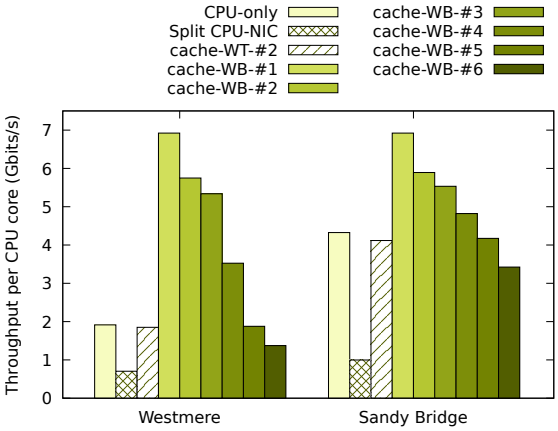


Figure 8: Throughput per CPU core of different implementations of the key-value store. WB = write-back, WT = write-through. #N in "cache-WB-#N" is the configuration number. Table 2 shows the cache sizes of the different configurations and their resulting hit rates.

Version (obtained from)	Effort (loc)	Details
Existing	1708	Hand-written C program
CPU-only (Existing)	replace 538 with 334	Refactor C program into FLOEM elements.
Split CPU-NIC (CPU-only)	add 296	Create queues. NIC remotely allocates items on CPU memory.
Caching (CPU-only)	add 43	Create a cache. Assign key, keylen, val, vallen.
NIC caching (Caching)	add 62	Create queues and segments.

Table 1: Effort to implement key-value store. The last column describes specific modification details other than creating, modifying, and rewiring elements. As a baseline, code relevant to communication on the CPU side alone was 240 lines in a manual C implementation.

Regarding performance, the third bar in Figure 8 reports the throughput when using a write-through cache with 2^{15} buckets and five entries per bucket, resulting in a 90.3% hit rate. According to the result, the write-through cache did not provide any benefit over the CPU-only design, even when the cache hit rate was quite high. Therefore, we configured the caching construct to use a write-back policy (by changing the cache policy parameter) because write-back generally yields higher throughput than write-through. The remaining bars show the performance when using a write-back cache with different cache sizes, resulting in the different hit rates shown

Config.	#1	#2	#3	#4	#5	#6	#2 (WT)
# of buckets	2^{15}	2^{15}	2^{15}	2^{15}	2^{14}	2^{14}	2^{15}
# of entries	∞	5	2	1	1	1	5
hit rate (%)	100	97.2	88.4	75.3	65.0	55.2	90.3

Table 2: The sizes of the cache (# of buckets and # of entries per bucket) on the NIC and the resulting cache hit rates when using the cache for the key-value store. All columns report the hit rates when using write-back policy except the last column for write-through. ∞ entries mean a linked list.

in Table 2. This offloading strategy improved throughput over the CPU-only design by 2.8–3.6× on Westmere and 28–60% on Sandy Bridge when the hit rate exceeded 88% (configuration #1–3).

Notice that at high cache hit rates, the throughput for this offload strategy was almost identical on Westmere and Sandy Bridge regardless of the CPU technology. The NIC essentially boosted performance on the Westmere server to be on par with the Sandy Bridge one. In other words, an effective NIC offload reduced the workload's dependency on CPU processing speed.

Case Study: Distributed Real-Time Data Analytics

Distributed real-time analytics is a widely-used application for analyzing frequently changing datasets. Apache Storm [1], a popular framework built for this task, employs multiple types of workers. Spout workers emit tuples from a data source; other workers consume tuples and may emit more tuples. A worker thread executes one worker. De-multiplexing threads route incoming tuples from the network to local workers. Multiplexing threads route tuples from local workers to other servers and perform simple flow control. Our specific workload ranked the top *n* users from a stream of Twitter tweets. In this case study, we optimized for throughput per CPU core. Figure 9 and Table 3 summarize the throughput and implementation effort of different strategies, respectively.

CPU-only: We ported demultiplexing, multiplexing, and DCCP flow-control from FlexStorm [23] into FLOEM but kept the original implementation of the workers as an external program. We used *callable segments* (Section 4.4) to define functions `inqueue_get` and `outqueue_put` for workers (in the external program) to obtain a task from the demultiplexer and send a task to the multiplexer (in FLOEM). This porting effort involved replacing 1,192 lines of code with only 350 lines. The code reduction here was much higher than in the key-value store application because FlexStorm's original implementation required many communication queues, which were replaced by FLOEM queues. The best CPU-only configuration that achieved the highest throughput per core used three cores for three workers (one spout, one counter, and one ranker), one core for demultiplexing, and two cores for multiplexing.

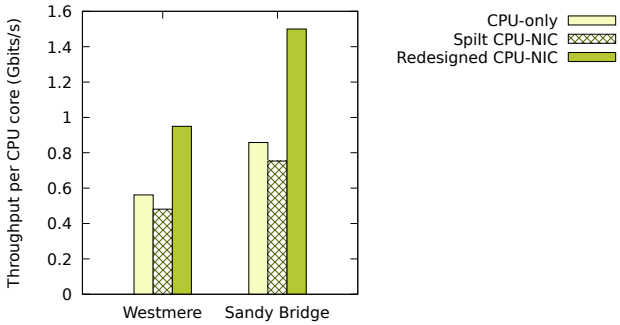


Figure 9: Throughput per CPU core of different Storm implementations

Version (obtained from)	Effort (loc)	Details
Existing	2935	Hand-written C program
CPU-only (Existing)	replace 1192 with 350	Refactor C program into FLOEMelements.
Split CPU-NIC (CPU-only)	modify 1	Change `device` parameter.
Redesigned (Split CPU-NIC)	add 23	Create bypass queues.

Table 3: Effort to implement Storm. The last column describes specific modification details other than creating, modifying, and rewiring elements.

Split CPU-NIC: As suggested in FlexNIC, we offloaded (de-)multiplexing and flow control to the NIC, by changing the device parameter (one line of code change). This version, however, lowered throughput slightly compared to the CPU-only version.

Redesigned CPU-NIC: The split CPU-NIC version can be optimized further. A worker can send its output tuple to another local worker or a remote worker over the network. For the former case, a worker sends a tuple to the multiplexer on the NIC, which in turn forwards it to the target worker on the CPU. Notice that this CPU-NIC-CPU round-trip is unnecessary. To eliminate this communication, we created bypass queues for workers to send tuples to other local workers without involving the multiplexer. With this slight modification (23 lines of code), we achieved 96% and 75% higher throughput than the CPU-only design on the Westmere and Sandy Bridge cluster, respectively.

Other Applications

The following three applications are common network function tasks. Because of their simplicity, we did not attempt to partition them across the CPU and NIC. Figure 10 reports throughput when using one CPU core on a Sandy Bridge server or offloading everything to the Cav-

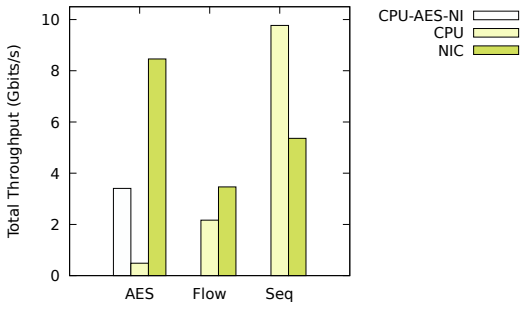


Figure 10: Throughput of AES encryption, flow classification, and network sequencer running on one CPU core and the LiquidIO NIC. 'CPU-AES-NI' uses AES-NI.

ium NIC. In our experiment, we used a packet size of 1024 bytes for encryption and network sequencer, and 80 bytes for flow classification.

Encryption is a compute-intensive stateless task, used for Internet Protocol Security. In particular, we implemented AES-CBC-128. We wrote two CPU versions: (1) using Intel Advanced Encryption Standard New Instructions (AES-NI), and (2) without AES-NI, which is available in only some processors. NIC Offloading improved throughput by 2.5× and 17.5× with and without AES-NI on CPU, respectively. Using AES-NI improved performance on the CPU but to a lesser degree than utilizing all encryption co-processors on the NIC. This result would be difficult to predict without an empirical test.

Flow classification is a stateful task that tracks flow statistics. We categorized flows using the header 5-tuple and used a probabilistic data structure (a count-min sketch) to track the number of bytes per flow. This application ran slightly faster on the NIC. Therefore, it seems reasonable to offload this task to the NIC if we want to spare CPU cycles for other applications.

Network sequencer orders packets based on predefined rules. It performs simple computation and maintains limited in-network state. This function has been used to accelerate distributed system consensus [29] and concurrency control [28]. Our network sequencer was 82% faster on the CPU core than on the NIC. Application throughput did not scale with the number of cores because of the group lock's contention; the number of locks acquired by each packet was 5 out of 10 on average in our synthetic workload, making this task inherently sequential. Therefore, using one fast CPU core yielded the best performance. We also tried running this program using multiple CPU cores, but throughput stayed the same as we increased the number of cores. On the NIC, using three cores offered the highest performance.

In summary, even for simple applications, it is not obvious whether offloading to the NIC improves or degrades performance. Using FLOEM lets us answer these questions quickly and precisely by simply changing the device parameter of the computation segment to either CPU or NIC. Comparing cost-performance or power-performance is beyond the scope of this paper. Nevertheless, one can use FLOEM to experiment with different configurations for a specific workload to optimize for a particular performance objective.

7.2 Logical-to-Physical Queue Mapping

Hypothesis 2 *Logical-to-physical queue mapping lets programmers implement packet steering, packet ordering, and different degrees of resource sharing.*

Packet steering. Storm, the second case study, required packet steering to the correct input queues, each dedicated to one worker. This was done by creating a queue with multiple physical instances and by setting `state.qid` according to an incoming tuple's type.

Packet ordering. The write-back cache implementation required in-order delivery between CPU and NIC to guarantee consistency (see Section 4.1).

Resource sharing. For the split NIC-CPU version of the key-value store, sending both GET and SET requests on separate physical queues offered 7% higher throughput than sharing the same queue. This is because we can use a smaller queue entry's size to transfer data for GET requests. In contrast, for our Storm application, sharing the same physical output queue between all workers yielded 16% higher throughput over separate dedicated physical queues. Since some workers infrequently produce output tuples, it was more efficient to combine tuples from all workers to send over one queue. Hence, it is difficult to predict whether sharing or no sharing is more efficient, so queue resource sharing must be tunable.

7.3 Inferred Data Transfer

Hypothesis 3 *Inferred data transfer improves performance relative to sending an entire packet.*

In this experiment, we evaluated the benefit of sending only a packet's live fields versus sending an entire packet over a queue. We measured the throughput of transmitting data over queues from the NIC to CPU when varying the ratio of the live portion to the entire packet's size (*live ratio*), detailed in Table 4. The sizes of live portions and packets were multiples of 64 bytes because performance was degraded when a queue entry's size was not a multiple of 64 bytes, the size of a CPU cache line. We used numbers of queues and cores that maximized throughput.

Live ratio	1/5	1/4	1/3	1/2	2/3	3/4	4/5
Live size (B)	64	64	64	64	128	192	256
Total size (B)	320	256	192	128	192	256	320
Speedup	3.1x	2.5x	2x	1.5x	1.3x	1.2x	1.2x

Table 4: Speedup when sending only the live portions when varying live ratios from a micro-benchmark. Sizes are in bytes (B).

As shown on the table, sending only live fields improved throughput by 1.2–3.1×. Additionally, we evaluated the effect of this optimization on the split CPU-NIC version of the end-to-end key-value store, whose queues from NIC to CPU transfer packets with a live ratio of 1/2. The optimization improved the throughput of this end-to-end application by 6.5%.

7.4 Queue Synchronization Layer

Hypothesis 4 *The queue synchronization layer enables high-throughput communication queues.*

We measured the throughput of three benchmarks. The first benchmark performed a simple packet forwarding from the NIC to CPU with no network activity, so its performance purely reflects the rate of data transfer over the PCIe bus rather than the rate of sending and receiving packets over the network. We used packet sizes of 32, 64, 128, and 256 bytes. The other two benchmarks were the write-back caching version of the key-value store and the redesigned CPU-NIC version of Storm.

Figure 11 displays the speedup when using the sync layer versus using primitive blocking DMA without batching (labeled "without sync layer"). The sync layer offered 9–15× speedup for pure data transfers in the first benchmark. Smaller packet sizes showed a higher speedup; this is because batching effectiveness increases with the number of packets in a batch. For end-to-end applications, we observed a 7.2–14.1× speedup for the key-value store and a 3.7× speedup for Storm. Note that the sync layer is always enabled in the other experiments. Hence, it is crucial for performance of our system.

7.5 Compiler Overhead

Hypothesis 5 *The* FLOEM *compiler has negligible overhead compared to hand-written code.*

We compared the throughput of code generated from our compiler to hand-optimized programs in C. To measure the compiler's overhead on the CPU, we ran a simple echo program, Storm, and key-value store. The C implementations of Storm and key-value store were taken from FlexStorm and one of FlexKVS's baselines [23]; these implementations are highly-optimized and perform better than the standard public implementations of Storm and memcached. On the NIC, we compared a simple

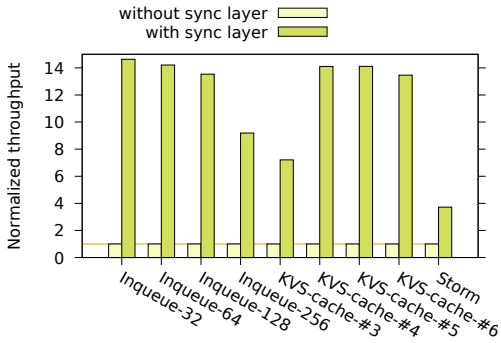


Figure 11: Effect of the queue synchronization layer. Throughput is normalized to that without the sync layer.

echo program, encryption, flow classification, and network sequencer. On average, the overhead was 9% and 1% on CPU and NIC, respectively. We hypothesize that the higher overhead on the CPU was primarily because we did not implement computation batching [24, 46], which was used for hand-optimized programs.

8 Discussion and Future Work

Multi-message packets. FLOEM can support a packet whose payload contains multiple requests via `Batcher` and `Debatcher` elements. Given one input packet, `Debatcher` invokes its one output port n times sequentially, where n is the number of requests in the payload. `Batcher` stores the first $n-1$ packets in its state. Upon receiving the last token, it sends out n packets as one value. The `Debatcher` element can inform the value of n to the `Batcher` element via the per-packet state. One can also take advantage of this feature to support computation batching, similar to Snap [46].

Multi-packet messages and TCP. Exploring the TCP offload with FLOEM is future work. FLOEM supports multi-packet messages via `Batcher` and `Debatcher` elements and could be used together with a TCP offload on the NIC, but our applications do not use TCP.

Shared data structures. In FLOEM, queues and caches are the only high-level abstractions for shared data structures between the NIC and CPU. However, advanced developers can use FLOEM to allocate a memory region on the CPU that the NIC can access via DMA operations, but they are responsible for synchronizing data and managing the memory by themselves.

Automation. Automatic program partitioning was among our initial goals, but we learned that it cannot be done entirely automatically. Different offloading strategies often require program refactoring by rewriting the graph and even graph elements. These program-specific

changes cannot be done automatically by semantics-preserving transformation rules. Therefore, we let programmers control the placement of elements while refactoring the program for a particular offload design. However, FLOEM would benefit from and integrate well with another layer of automation, like an autotuner or a runtime scheduler, that could select parameters for low-level choices (e.g., the number of physical queues, the number of cores, and the placement of each element) after an application has been refactored.

Other SmartNICs. The current FLOEM prototype targets Cavium LiquidIO but can be extensible to other SmartNICs that support C-like programming, such as Mellanox BlueField [32] and Netronome Agilio [6]. However, FPGAs [12, 33, 48] require compilation to a different execution model and the implementation of bodies of elements in a language compatible with the hardware.

9 Related Work

Packet processing frameworks. The FLOEM data-flow programming model is inspired by the Click modular router [34], a successful framework for programmable routers, where a network function is composed from reusable elements [34]. SMP Click [13] and RouteBricks [16] extend Click to exploit parallelism on a multi-processor system. Snap [46] and NBA [24] add GPU offloading abstractions to Click, while ClickNP [27] extends Click to support joint CPU-FPGA processing. Dragonet, a system for a network stack design, automatically offloads computations (described in data-flow graphs) to a NIC with fixed hardware functions rather than programmable cores [43, 44].

Other packet processing systems adopt different programming models. PacketShader [19] is among the first to leverage GPUs to accelerate packet processing in software routers. APUNet [17] identifies the PCIe bottleneck between the CPU and GPU and employs an integrated GPU in an APU platform as a packet processing accelerator. Domain-specific languages for data-plane algorithms, including P4 [10] and Domino [45], provide even more limited operations.

Overall, programming abstractions provided by existing packing processing frameworks are insufficient for our target domain, as discussed in Section 2.

Synchronous data-flow languages. Synchronous data-flow (SDF) is a data-flow programming model in which computing nodes have statically known input and output rates [25]. StreamIt [47] adopts SDF for programming efficient streaming applications on multicore architectures. Flextream [20] extends StreamIt with dynamic runtime adaptation for better resource utilization. More recently, Lime [21] provides a unified programming language based on SDF for programming heterogeneous computers that feature GPUs and FPGAs. Although some variations of these languages support dynamic input/output rates, they are designed primarily for static flows. As a result, they are not suitable for network applications, where the flow of a packet through a computing graph is highly dynamic.

Systems for heterogeneous computing. Researchers have extensively explored programming abstractions and systems for various application domains on various heterogeneous platforms [8, 11, 31, 35, 39, 41, 42]. FLOEM is unique among these systems because it is designed specifically for data-center network applications in a CPU-NIC environment. In particular, earlier systems were intended for non-streaming or large-grained streaming applications, whose unit of data in a stream (e.g., a matrix or submatrix) is much larger than a packet. Furthermore, most of these systems do not support a processing task that maintains state throughout a stream of data, which is necessary for our domain.

10 Conclusions

Developing NIC-accelerated network applications is exceptionally challenging. FLOEM aims to simplify the development of these applications by providing a unified framework to implement an application that is split across the CPU and NIC. It allows developers to quickly explore alternative offload designs by providing programming abstractions to place computation to devices; control mapping of logical queues to physical queues; access fields of a packet without manually marshaling it; cache application state on a NIC; and interface with an external program. Our case studies show that FLOEM simplifies the development of applications that take advantage of a programmable NIC, improving the key-value store's throughput by up to 3.6×.

Acknowledgments

This work is supported in part by MSR Fellowship, NSF Grants CCF–1337415, NSF ACI–1535191, NSF 16-606, and NSF 1518702, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, grants from DARPA FA8750–16–2–0032, by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) as well as gifts from Google, Intel, Mozilla, Nokia, Qualcomm, Facebook, and Huawei.

References

[1] Apache Storm. http://storm.apache.org. Accessed: 2017-11-15.

[2] Cavium Development Kits. http://www.cavium.com/octeon_software_develop_kit.html. Accessed: 2017-11-15.

[3] Cavium LiquidIO. http://www.cavium.com/LiquidIO_Adapters.html. Accessed: 2017-11-14.

[4] DPDK: Data Plane Develepment Kit. http://dpdk.org/. Accessed: 2017-11-07.

[5] IEEE P802.3bs 400 GbE Task Force. Adopted Timeline. http://www.ieee802.org/3/bs/timeline_3bs_0915.pdf. Accessed: 2017-11-16.

[6] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. Accessed: 2017-11-14.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[9] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, 2014.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[11] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[12] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, 2016.

[13] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 2001.

[14] Cisco. Introduction To RPC/XDR. http://www.cisco.com/c/en/us/td/docs/ios/sw_upgrades/interlink/r2_0/rpc_pr/rpintro.html. Accessed: 2018-09-07.

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, 2009.

[17] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[18] Google. Protocol Buffers. http://developers.google.com/protocol-buffers/. Accessed: 2018-09-07.

[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '10, 2010.

[20] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[21] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In

Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08, 2008.

[22] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[23] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[24] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, 2015.

[25] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[26] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[27] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '16, 2016.

[28] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[29] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016.

[30] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[31] C. K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, 2009.

[32] Mellanox Technologies. BlueField Multicore System on Chip. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[33] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[34] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99, 1999.

[35] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium Symposium on Operating Systems Principles*, SOSP '09, 2009.

[36] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016.

[37] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, 2014.

[38] P. M. Phothilimthana. *Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Sept 2018.

[39] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, 2014.

[41] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[42] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[43] P. Shinde, A. Kaufmann, K. Kourtis, and T. Roscoe. Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, 2013.

[44] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We Need to Talk About NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, 2013.

[45] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '16, 2016.

[46] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, 2013.

[47] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, 2002.

[48] N. Zilberman, Y. Audzevich, G. Kalogeridou, N. Manihatty-Bojan, J. Zhang, and A. Moore. NetFPGA: Rapid Prototyping of Networking Devices in Open Source. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

Graviton: Trusted Execution Environments on GPUs

Stavros Volos
Microsoft Research

Kapil Vaswani
Microsoft Research

Rodrigo Bruno
INESC-ID / IST, University of Lisbon

Abstract

We propose Graviton, an architecture for supporting trusted execution environments on GPUs. Graviton enables applications to offload security- and performance-sensitive kernels and data to a GPU, and execute kernels in isolation from other code running on the GPU and all software on the host, including the device driver, the operating system, and the hypervisor. Graviton can be integrated into existing GPUs with relatively low hardware complexity; all changes are restricted to peripheral components, such as the GPU's command processor, with no changes to existing CPUs, GPU cores, or the GPU's MMU and memory controller. We also propose extensions to the CUDA runtime for securely copying data and executing kernels on the GPU. We have implemented Graviton on off-the-shelf NVIDIA GPUs, using emulation for new hardware features. Our evaluation shows that overheads are low (17-33%) with encryption and decryption of traffic to and from the GPU being the main source of overheads.

1 Introduction

Recent trends such as the explosion in volume of data being collected and processed, declining yields from Moore's Law [16], growing use of cloud computing, and applications, such as deep learning have fueled the widespread use of accelerators such as GPUs, FPGAs [37], and TPUs [20]. In a few years, it is expected that a majority of compute cycles in public clouds will be contributed by accelerators.

At the same time, the increasing frequency and sophistication of data breaches has led to a realization that we need stronger security mechanisms to protect sensitive code and data. To address this concern, hardware manufacturers have started integrating trusted hardware in CPUs in the form of trusted execution environments (TEE). A TEE, such as Intel SGX [28] and ARM Trustzone [1], protects sensitive code and data from system administrators and from attackers who may exploit kernel vulnerabilities and control the entire software stack, including the operating system and the hypervisor. However, existing TEEs are restricted to CPUs and cannot be used in applications that offload computation to accelerators. This limitation gives rise to an undesirable trade-off between security and performance.

There are several reasons why adding TEE support to accelerators is challenging. With most accelerators, a device driver is responsible for managing device resources (e.g., device memory) and has complete control over the device. Furthermore, high-throughput accelerators (e.g., GPUs) achieve high performance by integrating a large number of cores, and using high bandwidth memory to satisfy their massive bandwidth requirements [4, 11]. Any major change in the cores, memory management unit, or the memory controller can result in unacceptably large overheads. For instance, providing memory confidentiality and integrity via an encryption engine and Merkle tree will significantly impact available memory capacity and bandwidth, already a precious commodity on accelerators. Similarly, enforcing memory isolation through SGX-like checks during address translation would severely under-utilize accelerators due to their sensitivity to address translation latency [35].

In this paper, we investigate the problem of supporting TEEs on GPUs. We characterize the attack surface of applications that offload computation to GPUs, and find that delegating resource management to a device driver creates a large attack surface [26, 36] leading to attacks as page aliasing that are hard to defend without hardware support. Interestingly, we also find that architectural differences between GPUs and CPUs *reduce* the attack surface in some dimensions. For instance, all recent server-class GPUs use 3D-IC designs with stacked memory connected to GPU cores via silicon interposers [4, 11]. Unlike off-package memory connected to the CPU using copper-based traces on the PCB, which are easy to snoop and tamper, it is extremely hard for an attacker to open a GPU package and snoop on the silicon interconnect between GPU and stacked memory, even with physical access to the GPU. Thus, it is a reasonable assumption to include on-package memory within the trust boundary.

Based on these insights, we propose Graviton, an architecture for supporting TEEs on GPUs. In Graviton, a TEE takes the form of a *secure context*, a collection of GPU resources (e.g., device memory, command queues, registers) that are cryptographically *bound* to a public/private key pair and isolated from untrusted software on the host (including the driver) and all other GPU

contexts. Graviton guarantees that once a secure context has been created, its resources can only be accessed by a user application/runtime in possession of the corresponding private key. As long as the key is protected from the adversary (e.g., the key is hosted in a CPU TEE), the adversary cannot access the context's address space. Graviton supports two additional primitives: *measurement* for generating remotely verifiable summaries of a context's state and the platform, and *secure memory allocation and deallocation* for letting a device driver dynamically allocate and free memory without compromising security.

Graviton achieves strong security by redefining the interface between the GPU driver and the hardware. Specifically, we prevent the driver from directly accessing security sensitive resources, such as page directories, page tables, and other memory containing sensitive code and data. *Instead, we require that the driver route all resource allocation requests through the GPU's command processor.* The command processor tracks ownership of resources, and ensures that no resource owned by a secure context can be accessed by the adversary. The command processor also ensures that the resources are correctly initialized on allocation to a secure context, and cleaned up on destruction, preventing attacks that exploit improper initialization [23, 36, 52].

Our design has several key attributes including low hardware complexity, low performance overheads and crypto-agility. Graviton requires no changes to the GPU cores, MMU, or the memory controller. All changes are limited to peripheral components, such as the GPU command processor and the PCIe control engine; this is largely due to the assumption that on-package memory can be trusted. Graviton places no restrictions on the instruction set available within the TEE. We also show that a GPU runtime can use Graviton to build secure versions of higher-level APIs, such as memory copy, kernel launch, and streams, which can be used to build applications with end-to-end confidentiality and integrity.

We have evaluated our design on NVIDIA Titan GPUs, gdev [21], an open-source CUDA runtime, and nouveau [29], an open source GPU driver. In the absence of hardware that implements the proposed extensions, we implement and emulate the extensions using interrupts delivered to the host. Our evaluation using a set of representative machine learning benchmarks suggests that the overheads of running compute-bound GPU applications using secure contexts are low (17-33%) for the level of security we provide. The overheads are dominated by the cost of authenticated encryption/decryption of kernel launch commands and user data.

In summary, we make the following contributions.

- We propose Graviton, an architecture for supporting TEEs on accelerators, such as GPUs. Graviton provides strong security properties even against an adversary that might control the entire software stack on the host, including the accelerator driver.
- We define a threat model that places trust in the GPU hardware, including on-package memory.
- We propose a minimal set of extensions to the GPU hardware for implementing Graviton and show how these extensions can be used to design applications with end-to-end security guarantees. The design requires no changes to the GPU cores, MMU, or memory controller, resulting in low hardware complexity and low performance overheads.

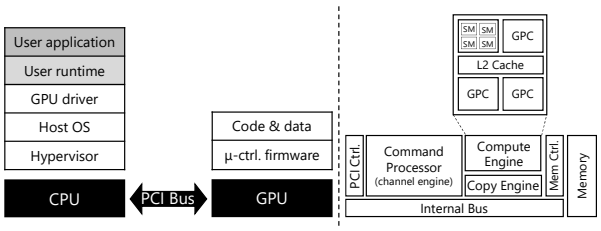


Figure 1: System stack (left) and hardware stack (right).

2 Background

2.1 GPU

We review the NVIDIA GPU architecture and the CUDA programming model to illustrate how a compute task is offloaded and executed on the GPU. We focus on the security critical parts of the architecture.

Software stack. A user-space application uses an API provided by the user-space GPU runtime (e.g., CUDA runtime), to program the GPU execution units with a piece of code known as the kernel, and transfer data between host and device memory. The GPU runtime converts each API call to a set of GPU commands for configuring the device and controlling kernel launches and data transfers. A GPU driver is responsible for submitting these commands to the GPU via the PCI bus and for managing device memory.

Hardware. The GPU (Figure 1) interfaces with the host CPU via the PCI control engine, which is connected with the rest of the GPU components via an internal bus. The key components are a command processor, compute and copy (DMA) engines, and the memory system, including the memory controller and memory chips.

The PCI control engine consists of (a) a PCI controller that receives incoming and outgoing PCI transactions, and (b) a master control engine, which exposes a set of memory-mapped-IO (MMIO) registers that are accessed by the host CPU to enable and disable the GPU engines.

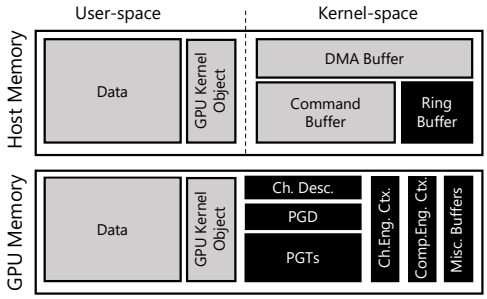


Figure 2: Host memory and GPU memory spaces.

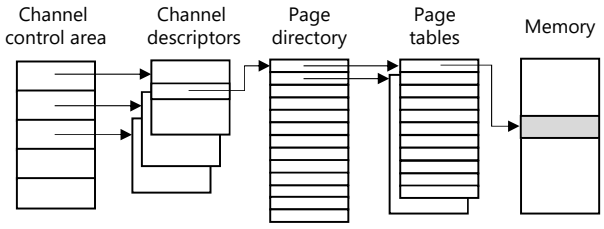


Figure 3: Channel-level address space management.

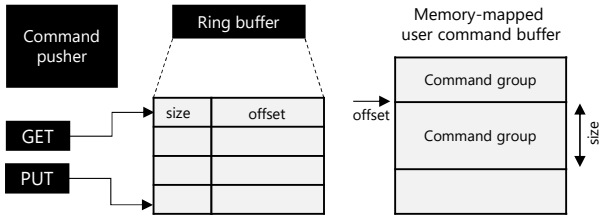


Figure 4: GPU command submission.

The command processor (aka channel engine) receives commands submitted by the device driver over as set of command queues known as *channels* and forwards them to the corresponding engines once they are idle. Channels are configured through a set of memory locations known as the *channel control area* which is mapped over the MMIO and serviced by the command processor.

The compute engine consists of a set of graph processing clusters (GPCs) and a shared L2 cache. Each GPC consists of a number of streaming multiprocessors (SMs), which are used to run GPU kernels. Each SM consists of multiple cores and a private memory hierarchy, including a read-only cache, L1 cache, and application-managed memory. GPU kernels specify the number of threads to be created, organized into thread blocks and grids. Thread blocks are divided into *warps*, where each warp is a unit of scheduling on each SM. Threads belonging to the same thread block share the caches and the application-managed memory.

Modern GPUs support virtual memory via a memory controller with page table walkers for address translation, and a hierarchy of TLBs. For example, in the NVIDIA Volta, the L1 cache is virtually addressed and the L2 is physically addressed. The GPU has a shared two-level TLB used while accessing the L2 cache [19].

Context and channel management. Execution on GPUs is context-based. A CUDA context represent the collection of resources and state (memory, data, etc.) that are required to execute a CUDA kernel. Resources are allocated to contexts to run a compute task and are freed when a context is destroyed. Each context has its own address space. GPUs use channels to isolate a context's address space from other contexts. A channel is the only way to submit commands to the GPU. Therefore, every GPU context allocates at least one GPU channel.

To create a channel, the device driver allocates a *channel descriptor* and multi-level page tables in device memory (Figure 2 and 3). For example, a simple two-level page table consists of the page directory (PGD) and a number of leaf page tables (PGT). The driver writes the channel descriptor address to the channel control area, and the page directory address in the channel descriptor. The page directory consists of entries that point to leaf page tables, and leaf page tables contain virtual-to-physical mappings. Page tables typically support small (4KB) and big pages (128KB). The device driver updates all these data structures over the PCI bus via BARs.

Once the channel has been created, the device driver allocates device memory for a few channel-specific data structures, including (a) the internal context of the channel and compute engines, (b) a fence buffer used for synchronization between the host CPU and GPU, and (c) an interrupt buffer for notifying the host with interrupts generated by the GPU engines.

Command submission. The command processor is responsible for fetching commands submitted by the software stack and relaying them to the appropriate GPU engines. Figure 4 shows the data structures used for command submission. The driver allocates two buffers in kernel space, a command and a ring buffer. The command buffer is memory-mapped to the user space. The runtime pushes groups of commands to the command buffer, updates the channel's ring buffer with the size and offset of each group, and then updates over MMIO a register called the PUT register with a pointer to the command group. When the PUT register is updated, the command processor fetches a command group from the buffers, and updates the GET register to notify the runtime that the commands have been fetched.

Programming model. Next, we present an overview of the main stages of dispatching kernels to the GPU.

Initialization. An application wishing to use the GPU first creates a CUDA context. During context creation,

the runtime allocates a DMA buffer for data transfers between host memory and device memory (Figure 2). Subsequently, the application loads one or more CUDA modules into the context. For each kernel defined in the module, the runtime creates a corresponding *kernel object* on the GPU by allocating device memory for (a) the kernel's code, (b) constant memory used by the kernel, and (c) local memory used by each thread associated with the kernel. The runtime then copies code and constant memory to device memory via DMA.

Memory allocation. The application allocates device memory for storing inputs and outputs of a kernel using a memory allocation API (`cudaMalloc` and `cudaFree`). Memory allocations are serviced by the driver, which updates the page directory and page tables.

Host-GPU transfers. When the application issues a host-to-device copy, the runtime pushes a command group to the context's channel, passing the virtual addresses of source and destination to the copy engine. Once the engine is configured, it translates virtual addresses to physical ones and initiates DMA transfers.

Kernel dispatch. When the application executes a kernel, the runtime passes a command group to the command processor that includes the kernel's context, the base address of the code segment, the entry program counter, the grid configuration, and the kernel's environment, which includes the stack and parameters values. The command processor uses these parameters to initialize compute engines, which in turn initialize and schedule the computation on GPU cores.

Sharing. A GPU can be used to execute multiple kernels from multiple host processes using techniques such as pre-emptive multi-tasking [33, 42], spatial multi-tasking [2, 34], simultaneous execution [47], multi-process service [30], or virtualization [25]. In such scenarios, it is the responsibility of the host (driver) to isolate kernels using the channel abstraction and virtual memory. While spatial multi-tasking advocates for SM partitioning, it still shares memory resources and relies on virtual memory for isolation. Even in devices that support SR-IOV and partition resources in hardware (e.g., AMD MxGPU), system software is still responsible for assigning virtual devices to virtual machines.

2.2 Intel SGX

Trusted execution environments, or *enclaves* (e.g., Intel SGX) protect code and data from all other software in a system. With OS support, an untrusted hosting application can create an enclave in its virtual address space. Once an enclave has been initialized, code and data within the enclave is isolated from the rest of the system, including privileged software.

Intel SGX enforces isolation by storing enclave code and data in a data structure called the Enclave Page Cache (EPC), which resides in a pre-configured portion of DRAM called the Processor Reserved Memory (PRM). The processor ensures that any software outside the enclave cannot access the PRM. However, code hosted inside an enclave can access both non-PRM memory and PRM memory that belongs to the enclave. SGX includes a memory encryption engine that encrypts and authenticates enclave data evicted to memory, and ensures integrity and freshness.

In addition to isolation, enclaves also support *remote attestation*. Remote attestation allows a remote challenger to establish trust in an enclave. In Intel SGX, code hosted in an enclave can request a *quote*, which contains a number of enclave attributes including a measurement of the enclave's initial state. The quote is signed by a processor-specific attestation key. A remote challenger can use Intel's attestation verification service to verify that a given quote has been signed by a valid attestation key. The challenger can also verify that the enclave has been initialized in an expected state. Once an enclave has been verified, the challenger can set up a secure channel with the enclave (using a secure key exchange protocol) and provision secrets such as encrypted code or data encryption keys to the enclave.

3 Threat Model

We consider a strong adversary who controls the entire system software, including the device drivers, the guest operating system, and the hypervisor, and has physical access to all server hardware, including the GPU. Clearly, such an adversary can read and tamper with code or data of any victim process. The adversary can also access or tamper with user data in DMA buffers or with commands submitted by the victim application to the GPU. This gives the adversary control over attributes, such as the address of kernels being executed and parameters passed to the kernel. The adversary may also access device memory directly over MMIO, or map a user's GPU context memory space to a channel controlled by the adversary. In multi-tasking GPUs, malicious kernels can be dispatched to the GPU, thereby accessing memory belonging to a victim's context. These attacks are possible even in a virtualized environment (e.g., even if a device supports SR-IOV) because the mapping between VMs and virtual devices is controlled by the hypervisor.

An adversary with physical access to the server can mount snooping attacks on the host memory bus and the PCIe bus. However, we do trust the GPU and CPU packages and firmware, and assume that the adversary cannot extract secrets or corrupt state within the packages. This implies that we trust CPUs to protect code and data hosted inside TEEs. Side-channel attacks (e.g., based on speculative execution, access patterns and timing) and

denial-of-service attacks are also outside the scope of this paper. Side channels are a serious concern with trusted hardware [10, 12, 22, 38, 45, 50] and building efficient counter measures remains an open problem. In Graviton, we use TEEs to host the user application and the GPU runtime.

Unlike host memory, which is untrusted, we trust on-package GPU memory as GPU cores are attached to memory using silicon interposers, which make it extremely difficult for an attacker to mount snooping or tampering attacks. There is an emerging class of attacks on stacked integrated circuits (ICs), such as attacks where the package assembler inserts a trojan die between the GPU and memory dies [49]. Developing mitigations for these attacks is ongoing work [3] and outside the scope of this paper.

Even under this threat model, we wish to guarantee confidentiality and integrity for applications that use GPUs. Specifically, we wish to guarantee that the adversary cannot observe or tamper with code, data, and commands transferred to/from the GPU by a trusted application that runs in a CPU TEE or an on-premise machine. Finally, we wish to guarantee that the GPU computation proceeds without interference from the adversary.

4 Overview

Consider a CUDA application (Figure 5) that performs matrix multiplication, which is a key building block in machine learning algorithms. The application creates a new CUDA context (implicitly on the first CUDA API call), allocates memory for input and output matrices in host and device memory, populates the matrices, and then invokes the matrix multiplication kernel on the GPU (not shown), passing pointers to device memory and other kernel's parameters. After the kernel has completed, the application copies the results into host memory and releases memory allocated on the GPU.

As described earlier, an attacker with privileged access to the server can easily recover the contents of the matrices and the result even if this application is hosted in a CPU enclave. We can harden this application against such attacks simply by linking it against Graviton's version of the CUDA runtime. Graviton's version of the runtime creates a secure context (instead of a default context) on a Graviton-enabled GPU. In this process, the runtime authenticates the GPU and establishes a secure session with the GPU's command processor, with session keys stored in CPU enclave memory.

The runtime also provides a custom implementation of `cudaMalloc`, which invokes the device driver to allocate GPU memory and additionally verifies that allocated memory is not accessible from any other context or from the host. The secure implementation of `cudaMemcpy` ensures that all transfers between host and the GPU, including code and data, are encrypted and authenticated using keys inaccessible to the attacker. The implementation of `cudaLaunch` sends encrypted launch commands to the GPU's command processor over a secure session. Finally, the implementation of `cudaFree` authorizes the GPU's command processor to unmap previously allocated pages from page tables, and scrubs their content, enabling the driver to reuse the pages without leaking sensitive data.

```
int main() {
  ...
  float* h_A = malloc(M*N*sizeof(float));
  float* h_B = malloc(N*K*sizeof(float));
  float* h_C = malloc(M*K*sizeof(float));
  float* d_A, d_B, d_C;
  ...
  cudaMalloc((void**)&d_A, M*N*sizeof(float)
      );
  ...
  populate_matrices(h_A, h_B);
  cudaMemcpy(d_A, h_A, M*N*sizeof(float),
      cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, N*K*sizeof(float),
      cudaMemcpyHostToDevice);
  ...
  matrixMul<<<grid,threads>>>(d_C,d_A,d_B,M,
      N,K);
  ...
  cudaMemcpy(d_C, h_C, M*K*sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  ...
}
```

Figure 5: Sample CUDA application.

5 Graviton Architecture

In this section, we describe extensions to existing GPU architectures for supporting secure contexts.

5.1 Remote Attestation

A Graviton-enabled GPU supports remote attestation for establishing trust between a secure context and a remote challenger. Hardware support for attestation is similar to TPMs; we require (a) a secret, known as the root endorsement key (EK), to be burned into the device's e-fuses during manufacturing and (b) a cryptographic engine for asymmetric key generation and signing. The EK is the root of trust for attestation and never leaves the GPU package. During boot, the GPU generates a fresh attestation key (AK) pair and stores the private key securely within the command processor. The GPU also signs the public part of the AK with the EK and makes it available to the device driver, which in turn sends the

signed AK to a trusted CA. The CA validates the signature using a repository of public endorsement keys provisioned by the manufacturer and generates a signed AK certificate. The certificate is stored by the device driver and used during secure context creation to prove to a challenger that the GPU holds and protects the private attestation key.

5.2 Secure Context Management

In Graviton, a secure context consists of one or more *secure channels*. We extend the GPU's command processor with new commands for creation, management, and destruction of secure channels (Figure 6).

A secure channel is created using the command `CH_CREATE`, which requires as parameters a channel identifier and a public key UK_{pub}. On receiving the request, the command processor generates a fresh channel encryption key (CEK) for encrypting commands posted to this channel. The public key UK_{pub}, CEK, and a counter are stored in a region of device memory accessible only to the command processor. `CH_CREATE` may be used to create multiple channels associated with the same secure context by passing the same UK_{pub}, in which case all such channels will use the same CEK.

After generating the CEK, the command processor establishes a *session* by securely transferring the CEK to the trusted user-space runtime. The command processor encrypts the CEK with UK_{pub} and generates a *quote* containing the encrypted CEK and a hash of UK_{pub}. The quote contains the channel identifier and security critical platform-specific attributes, such as the firmware version, and flags indicating whether preemption and debugging are enabled. The quote is signed using AK. The device driver passes this quote and the AK certificate (obtained during initialization) to the user-space runtime. The runtime authenticates the response by (a) verifying the AK certificate, (b) verifying the quote using the public AK embedded in the certificate, and (c) checking that the public key in the quote matches UK_{pub}. The runtime can then decrypt the CEK and use it for encrypting all commands sent to the GPU.

Once a session has been established, the command processor authenticates and decrypts all commands it receives over the channel using the CEK. This guarantees that only the user in possession of the CEK can execute tasks that access the context's address space. We use authenticated encryption (AES in GCM mode) and the per-channel counter as IV to protect commands from dropping, replay, and re-ordering attacks. This ensures that all commands generated by the GPU runtime are delivered to the command processor without tampering.

5.3 Secure Context Isolation

In existing GPUs, the responsibility of managing resources (e.g., device memory) lies with the device driver. For example, when allocating memory for an application object, the driver determines the virtual address at which to allocate the object, then determines physical pages to map to the virtual pages, and finally updates virtual-physical mappings in the channel's page tables over MMIO. This mechanism creates a large attack vector. A compromised driver can easily violate channel-level isolation—e.g., by mapping a victim's page to the address space of a malicious channel.

One way of preventing such attacks and achieving isolation is to statically partition resources in hardware between channels. However, this will lead to under-utilization of resources. Moreover, it prohibits low-cost sharing of resources between channels, which is required to implement features, such as streams. Instead, Graviton guarantees isolation by imposing a strict *ownership* discipline over resources in hardware, while allowing the driver to dynamically partition resources.

More formally, consider a physical page P that is mapped to a secure channel associated with a secure context C and a channel encryption key CEK, and contains sensitive data. We consider any object (code and data) allocated by the application in a secure context and all address space management structures (i.e., channel descriptor, page directory and page tables) of all channels as sensitive. We propose hardware changes to a GPU that enforce the following invariants, which together imply isolation.

Invariant 5.1 *P cannot be mapped to a channel associated with a context $C' \neq C$.*

Invariant 5.2 *P cannot be unmapped without authorization from the user in possession of CEK.*

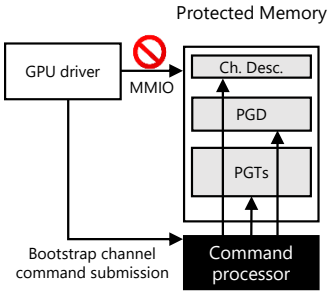
Invariant 5.3 *P is not accessible over MMIO to untrusted software on the host CPU.*

Invariant 5.4 *P is cleared before being mapped to another channel associated with a context $C' \neq C$.*

In the rest of this section, we describe hardware extensions for enforcing these invariants, and discuss their implications on the driver-GPU interface.

Memory regions. Our first extension is to partition device memory into three regions: *unprotected*, *protected* and *hidden*, each with different access permissions.

The *unprotected* region is a region in memory that is both visible from and accessible by the host via PCI BAR



Command	Attributes	Description
CH_CREATE	chid, pgd_address, public_key → quote	Sets page directory address within channel descriptor Returns quote for secure channels
CH_DESTROY	chid, MAC	Unsets channel descriptor and page directory Expects a keyed MAC of command for secure channels
CH_PDE	chid, pgd_index, PDE	Sets page table address in a page directory entry
CH_PTE	chid, pgd_index, pgt_index, size, PTE[], MAC → summary, MAC	Sets VA-PA mapping in page table entry Expects keyed MAC for user-authorized deallocation Returns a summary along with a MAC for secure allocations
CH_MEASURE	chid, address, size → measurement, MAC	Returns a measurement of a range of virtual addresses mapped in a secure channel along with a MAC

Figure 6: Commands for configuring a channel's address space and measuring the address space. PDE and PTE refer to page directory and page table entries respectively, and a MAC is a keyed message authentication code.

registers. The driver can use this region to allocate non-sensitive memory objects (e.g., synchronization and interrupt buffers) that are accessed over MMIO. This region can also be accessed from the GPU engines.

The *protected* region is visible to but not accessible from the host. The driver can allocate objects within the region (by creating page mappings), but cannot access the region directly over MMIO. Thus, this region can only be accessed from the GPU engines.

The *hidden* region is not visible or accessible to host CPU or to the GPU engines. Memory in this region are not accessible over PCI and are not mapped into any channel's virtual address space. This region is exclusively reserved for use by the command processor for maintaining metadata, such as ownership state of protected memory pages and per-channel encryption keys.

Regions can be implemented using simple range checks on MMIO accesses in the PCI controller and on commands that update address-space management structures in the command processor. The size of each region can be configured during initialization by untrusted host software. The size does not affect security; only availability as the system administrator could block creation of secure contexts by allocating a small protected region.

Address-space management. The next set of extensions are designed to enforce Invariant 5.1 and Invariant 5.2. We achieve this by decoupling the task of allocating and deallocating virtual and physical memory from the task of managing device-memory-resident address translation data structures (i.e., page directories and page tables) and delegating the latter to the GPU's command processor. In particular, we allow the driver to decide *where* in virtual and physical memory an object will reside, but require that the driver route requests to update page directories and page tables through the command processor using the API described in Figure 6.

The implementation of the API in the command processor enforces these invariants by tracking *ownership* of physical pages in the protected region in a data structure called the *Protected Memory Metadata* (PMM). We first describe PMM and then the commands.

Tracking ownership. The PMM is a data structure located in hidden memory, making it invisible to the host. It is indexed using the physical address of a memory page. Pages are tracked at the granularity of a small page (i.e., 4 KB). The PMM maintains the following attributes for each physical page.

- The attribute *owner_id* is the identifier of the channel that owns the page.
- The attribute $state \in \{\texttt{FREE}, \texttt{MAPPED}\}$ represents whether the page is free or already mapped to some channel. The initial value is `FREE`.
- The attribute `refcnt` tracks the number of channels a physical page has been mapped to.
- The attribute $lock \in \{\texttt{UNLOCKED}, \texttt{LOCKED}\}$ represents whether the page requires explicit authorization to be unmapped.
- The attribute *pgd_index* is an index into the page directory that points to the page table containing the mapping for the current page. Using this attribute, the command processor can reconstruct the virtual address of a physical page. In that sense, the PMM acts as an inverted page table for the protected region—i.e., stores $PA \rightarrow VA$ mappings.
- The attribute *pgt_entrycnt* is a 2-byte value that tracks the number of pages table entries allocated within a page table. Using this attribute, the command processor can know if a locked page table is empty and hence may be unmapped.

Assuming each PMM entry requires 64-bits, the total size of the PMM for a GPU with 6GB of physical memory is 12MB, which is $\sim 0.2\%$ of total memory.

Commands. The new commands for context and address space management use the PMM to enforce Invariant 5.1 and Invariant 5.2 as follows:

`CH_CREATE`**.** This command takes as a parameter the address of the page directory (*pgd_address*) for the newly created channel *chid*. It checks whether the channel descriptor and page directory are allocated on pages in the protected region, and that the pages are `FREE`. The former constraint ensures that after channel creation, the driver does not bypass the API and access the channel descriptor and page directory directly over MMIO.

If the checks succeed, the pages become `MAPPED` and the *owner_id* attribute of the pages is updated to the identifier of the channel being created. If a secure channel is being created (using a public key), the pages become `LOCKED`. The command processor then updates the address of the page directory in the channel descriptor, and clears the contents of pages storing the page directory to prevent an attacker from injecting stale translations. `CH_CREATE` fails if any of the pages containing the channel descriptor or the page directory is already `LOCKED` or `MAPPED` to an existing channel.

`CH_PDE`**.** This command unmaps an existing page table if one exists and maps a new page table at the index *pgd_index* in the page directory of the channel.

Before unmapping, the command checks if the physical pages of the page table are `UNLOCKED` or the *pgt_entrycnt* attribute is zero. In either case, the command decrements `refcnt`. If `refcnt` reaches zero, the pages become `FREE`. The command fails if the driver attempts to unmap a `LOCKED` page table or a page table with valid entries.

Before mapping a new page table, the command checks whether the page table is allocated on `FREE` pages in the protected region. If the checks succeed, the pages become `MAPPED`. Additionally, if the channel is secure, the pages become `LOCKED`. However, if these pages are already `MAPPED`, the command checks if the channel that owns the page (the current *owner_id*) and the channel that the page table is being mapped to belong to the same context by comparing the corresponding public key hashes. If the hashes match, the page's reference count is incremented. This allows physical page tables and hence physical pages to be shared between channels as long as they share the same context; this is required for supporting features such as CUDA streams [31]. If either of the checks succeed, the command creates a new entry in the page directory and clears the contents of the pages storing the page table. The command fails if the page table is mapped to a channel associated with a different context.

`CH_PTE`**.** This command removes existing mappings and creates new mappings (specified by PTE) for a contiguous range of virtual addresses of size *size* starting at *VA*, where *VA* is the virtual address mapped at index *pgt_index* by the page table at index *pgd_index* in the channel *chid*'s page directory. Before clearing existing page table entries, the command checks if the physical pages are `LOCKED`. To remove mappings for `LOCKED` physical pages, the command requires explicit authorization by the user runtime in the form of a MAC over the tuple $\{chid, VA, size\}$ using the CEK and a per-channel counter as the initialization vector (IV). The unforgeability of the MAC coupled with the use of a counter for IV ensures that a malicious driver cannot forge a command that unmaps physical pages allocated to secure channels, and then remapping them to other channels. If the checks and MAC verification succeed, the pages transition to `FREE`, and the page table entries are cleared.

Similarly, before creating new mappings, the command checks if the pages are `FREE`. Additionally, if the channel is secure, the command checks if the pages are located in the protected region (for sensitive code and data, discussed in Section 6). If the checks succeed, the page become `MAPPED` and if the page is being mapped to a secure channel, the pages become `LOCKED`.[1] If pages are already `MAPPED`, the command checks if the channel that owns the page (the current *owner_id*) and the channel that the page is being mapped to belong to the same context by comparing the corresponding public key hashes. On success, the command increments the *pgt_entrycnt* of the page table, updates the page table, and issues a TLB flush to remove any stale translations. While conventionally the latter is the responsibility of the device driver, in our design, the flush is implicit. The command fails if any of the pages are mapped to a channel associated with a different context.

When the command succeeds, it generates a *summary* structure, which encodes all $VA \rightarrow PA$ mappings created during the invocation of `CH_PTE`. The summary is a tuple $\{chid, VA, n, k, HASH(p_1, ..., p_n)\}$, where VA is the starting virtual address of the memory being allocated, n is the number of pages allocated in the protected region, k is the number of total pages allocated, and $p_1, ...p_n$ are addresses of protected physical pages. The command processor also generates a keyed MAC over this summary using the CEK. As described later, this summary is used by the runtime to verify that sensitive code and data are allocated in protected region.

`CH_DESTROY`**.** This command frees memory allocated to a channel by walking the page directory, finding physical pages owned by the channel, and resetting their en-

[1] Note that `CH_PTE` also permits pages in the unprotected region to be mapped to a secure channel; these pages can be accessed over MMIO and are used to store objects, such as fence buffers required by the driver for synchronization.

tries in the PMM. It then unmaps physical pages of the channel descriptor and the page directory, decrements `refcnt` for pages used for page tables, and pages become `FREE` if their `refcnt` reduces to 0.

For secure channels, the command requires explicit authorization in the form of a MAC over *chid* using the CEK and a per-channel counter as IV. However, the command processor also accepts unauthorized instances of this command; this enables the device driver to reclaim resources in scenarios where the user runtime is no longer able to issue an authorized command—e.g., due to a process crash.

In such a scenario, the command processor walks PMM to find physical pages mapped exclusively to the channel's address space, unmaps them, decrements their `refcnt` and clears their contents if their `refcnt` reduces to 0. The command processor also flushes all caches because memory accesses of the command processor do not go through the memory hierarchy of compute engines.

A malicious driver may misuse this mechanism to reclaim resources of a channel which is still in use, resulting in denial of service; there is not violation of confidentiality or integrity as pages containing sensitive information, including the channel descriptor, are scrubbed.

`CH_MEASURE` We extend the command processor with a command `CH_MEASURE` for generating a verifiable artifact that summarizes the contents of a secure channel. The artifact can be used to prove to a challenger (e.g., a GPU runtime) that a channel exists in a certain state on hardware that guarantees channel isolation. In our implementation, `CH_MEASURE` takes as parameters a range of virtual pages that should be included in the measurement. It generates a *measurement*, which contains a digest (HMAC) of the content of pages within the requested range, and a keyed MAC of the measurement using the CEK.

Bootstrapping. Introducing a command-based API for address-space management raises the following issue: *How does the driver send commands for managing the address space of secure channels without having access to the channel-specific CEK?* We overcome this by requiring the driver to use separate channels, which we refer to as *bootstrap channels*, for routing address-space management commands for all other channels. We allow the driver to create and configure one or more bootstrap channels over MMIO and allocate their address-space management structures in the unprotected region.

The command processor identifies a channel as a bootstrap channel by intercepting MMIO writes to the channel descriptor attribute in the channel control area. If the address being written to this attribute is in the unprotected region, the corresponding channel is marked as a bootstrap channel.

To ensure that the driver does not use a bootstrap channel to violate isolation of secure channels, the command processor prohibits a bootstrap channel from issuing commands to the copy and compute engines since such commands can be used to access sensitive state. The command processor also checks that all commands executed from a bootstrap channel are used to configure non-bootstrap channels. This prevents an adversary from allocating protected memory pages of a secure context as page directory and/or page tables for a bootstrap channel and then leveraging the `CH_PDE` and `CH_PTE` commands to tamper with the memory of the secure context.

Big page support. The virtual memory subsystem on modern GPUs employ multiple page sizes. For example, in NVIDIA GPUs, each page directory entry consists of two entries, one pointing to a small page (4KB) table, and another to a big page (128KB) table. Our design requires minor extensions to support large pages. In the PMM, we continue to track page metadata at the small page granularity, but we add a bit to each entry to indicate if the corresponding physical page was mapped to a small or big virtual page. In addition, we require an additional parameter in the `CH_PDE` and `CH_PTE` commands to specify whether the updates are to a small or big page table. Finally, these commands check that the same virtual page is not mapped to two different physical pages.

Error handling. When a command fails, the command processor writes the error in an SRAM register that is accessible by the device driver over MMIO. This allows the device driver to take necessary actions so as to guarantee consistent view of a channel's address space between the command processor and the device driver.

6 Software Stack

We now describe new CUDA primitives supported by the Graviton runtime that use secure contexts and enable design of applications with strong security properties.

Secure memory management. The Graviton runtime supports two primitives `cudaSecureMalloc` and `cudaSecureFree` for allocating and deallocating device memory in the protected region.

`cudaSecureMalloc` guarantees that allocated GPU memory is owned by the current context (Invariant 5.1) and lies within the protected region (Invariant 5.3). Much like the implementation of `cudaMalloc`, `cudaSecureMalloc` relies on the device driver to identify unused virtual memory and physical pages in device memory, and update page directory and page tables using the commands `CH_PDE` and `CH_PTE`. As described above, these commands implement checks to enforce Invariant 5.1. The runtime enforces Invariant 5.3 using the

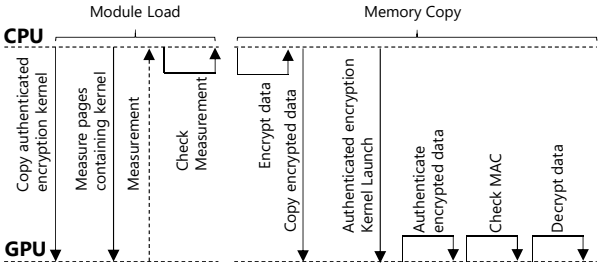


Figure 7: Secure memory copy protocol. The kernel is copied to the GPU during module creation.

summary structure generated by the `CH_PTE` command. In particular, the runtime uses the CEK and channel-specific counter to authenticate the summary structure(s) returned by the driver. The driver may return multiple summary structures in case the allocation spans multiple page tables. After authentication, the runtime can verify that memory objects are allocated in protected region using the attribute *n* in the summary.

`cudaSecureFree` first clears all allocated pages using a memset kernel. It then generates a MAC over the starting virtual address and size of the object using the CEK, and passes the MAC to the driver, which generates `CH_PTE` commands to remove entries from the page table. The MAC serves as an authorization to remove entries from the page table (Invariant 5.2). In the case where an object spans multiple page tables, the runtime generates one MAC per page table.

An implication of the redefined interface between the driver and the hardware is the inability of the driver to compact pages allocated to secure channels. Conventionally, the driver is responsible for compacting live objects and reducing fragmentation in the physical address space. However, Graviton prohibits the driver from accessing these objects. This can cause fragmentation in the protected region. We leave hardware support for compaction for future work.

Secure memory copy. The Graviton runtime supports the primitive `cudaSecureMemcpy` for securely copying code and data from the host TEE to device memory and vice versa. The protocol (Figure 7) works as follows.

1. After a secure context is created, the runtime allocates device memory using `cudaSecureMalloc` and copies a kernel that performs authenticated decryption (in clear text) into allocated memory, referred to as *AuthDec*. To ensure that the kernel was copied correctly without tampering, the runtime measures the region in device memory that contains the kernel (using `CH_MEASURE`), and checks the the returned digest matches the digest of the kernel computed inside the host TEE.
2. The implementation of `cudaSecureMemcpy` first encrypts data to be copied using a fresh symmetric key within the host TEE, and copies encrypted data to untrusted memory.
3. The runtime initiates a DMA to transfer encrypted data to target memory region. The command group that initiates the DMA is encrypted and integrity protected using the CEK.
4. The runtime uses the *AuthDec* kernel to decrypt data in device memory. It issues a command group to launch the kernel, passing the data's virtual address, the data encryption key, and the expected authentication tag as the kernel's parameters.
5. *AuthDec* authenticates encrypted data and generates an authentication tag which is checked against the expected authentication tag. If the check succeeds, the kernel decrypts the data in device memory, overwriting the encrypted data in the process.

A key attribute of secure memory copy is crypto-agility. Since the primitive is implemented fully in software, the runtime may support various encryption and authentication schemes without hardware changes.

Secure kernel launch. `cudaSecureKernelLaunch` uses secure memory copy to transfer the kernel's code and constant memory to the GPU, and then issues a command group to launch the kernel.

Recent GPUs have introduced preemption at instruction and/or thread-block boundaries. Extending our design to support preemption at the boundary of thread blocks is relatively straightforward because thread blocks are independent units of computation [42] and all ephemeral state, such as registers, application-managed memory, caches and TLBs can be flushed on preemption. Instruction-level preemption can also be supported by saving and restoring ephemeral state to and from a part of hidden memory reserved for each channel.

Secure streams. CUDA streams is a primitive used to overlap host and GPU computation, and I/O transfers. Each stream is assigned a separate channel, with each channel sharing the same address space, to enable concurrent and asynchronous submission of independent tasks. Our design supports secure streams (`cudaSecureStreamCreate`) by allowing channels within the same context to share page tables and pages. In particular, the runtime can remap a memory object to the stream's address space. Much like allocation requests, the driver uses `CH_PTE` command to update page tables. The runtime verifies that the $HASH(p_1, ..., p_n)$ generated by the `CH_PTE` command matches with the hash of the requested memory object.

7 Evaluation

7.1 Implementation

We implemented Graviton using an open-source GPU stack consisting of *ocelot*, an implementation of the CUDA runtime API [14], *gdev*, which implements the CUDA driver API [21], and *libdrm* and *nouveau*, which implement the user- and kernel-space GPU device driver [29]. Due to gdev's limitations (e.g., inability to use textures), we could not use some operations in cuBLAS (NVIDIA's linear algebra library) such as matrix-matrix (GEMM) and matrix-vector multiply (GEMV). Instead, we used implementations from Magma, an open-source implementation of cuBLAS with competitive performance [43]. Our implementation does not yet use SGX for hosting the user application and GPU runtime—porting the stack to SGX can be achieved using SGX-specific containers [5, 39, 44], and is outside the scope of this work.

Command-based API emulation. Since command processors in modern GPUs are not programmable, we *emulate* the proposed commands in software. Our emulator consists of (a) a runtime component which translates each new command and its parameters into a sequence of existing commands that triggers interrupts during execution, and (b) a kernel component, which handles interrupts, reconstructs the original command in the interrupt handler, and implements the command's semantics. The emulator uses the following commands: `REFCNT` sets the value of a 32-bit register in the channel control area which is readable from the host, `SERIALIZE` waits until previous commands are executed, `NOP` and `NOTIFY` triggers an interrupt when a subsequent `NOP` command completes.

Figure 8 shows the pseudo-code of the emulator along with an example for the `CH_CREATE` command. When a command is submitted, the runtime invokes the function *cmd_emu*, which translates each 32-bit value *v* in the command's original bit stream into the following sequence of commands: `REFCNT` with *v* as the parameter, `SERIALIZE`, `NOTIFY`, and `NOP`. This sequence is pushed into the ring buffer (using *push_ring_emu*), from where it is read by the command processor. When the command processor executes this sequence, it raises an interrupt, and an interrupt handler (*interrupt_handler*) is called on the host. The handler implements a state machine that reads the register in the channel control area and reconstructs the original command one value at a time. After reconstructing the entire command, the emulator implements its semantics (in this case using *chcreate_emu*) using reads and writes to device memory over MMIO (not shown in the figure).

We choose this emulation strategy because it allows us to run the software stack *as if* we had hardware support. Furthermore, it gives us a conservative approximation of performance because every command processor access to the PMM and memory mapping data structures in device memory translates into an access from the host to device memory over PCIe.

```
u32* ring_buf; /* ring buffer for context */
void push_ring(val) {
  *ring_buf=val;
  ring_buf++;
}

void push_ring_emu(u32 value) {
  push_ring(REF_CNT);
  push_ring(value);
  push_ring(SERIALIZE);
  push_ring(interrupt_buffer_addr >> 32);
  push_ring(interrupt_buffer_addr);
  push_ring(NOTIFY);
  push_ring(NOP);
}

void cmd_emu(u32 cmd, u32 param[], u32 size)
     {
  ...
  push_ring_emu(cmd);
  for (int i=0; i<size; i++)
    push_ring_emu(param[i]);
  ...
}

u32 chcreate(ctx_t *ctx, void *chan_base,
    void *pgd, u8 *pub_key) {
  ...
  cmd_emu(CMD_CHCREATE, param, 6);
  ...
}

void interrupt_handler(device_t *dev) {
  ...
  u32 val = mmio_rd32(dev);
  if (val != 0 || dev->cmd != 0) {
    if (dev->cmd == 0)
      dev->cmd = val;
    else {
      dev->param[dev->size++] = val;
      switch (dev->cmd) {
          ...
          case CMD_CHCREATE: {
            if (dev->size == 6) {
              chcreate_emu(dev);
              dev->size = 0;
              dev->cmd = 0;
            }
          } break;
  ...
}
```

Figure 8: Pseudo code for command emulation.

Command group authentication emulation. We also emulate the command processor logic for authenticated decryption of command groups. Our emulator intercepts encrypted command groups *before* they are copied to de-

vice memory and decrypts them. As we show later, this gives us a conservative approximation of performance since we decrypt command groups in software (aided by AES-NI) instead of a hardware encryption engine.

To estimate performance that can be achieved using a hardware encryption engine, we added a mode in our emulator which encrypts command groups on the host, but sends command groups in cleartext to the device, and adds a delay equal to the latency of decrypting the command group using a hardware engine. We compute the latency of decryption using the published latency of decrypting a block in hardware [40] and the size of the command group (in blocks).

Secure memory copy. Our implementation of secure memory copy combines AES and SHA-3 for authenticated encryption. We choose SHA-3 as its parallel tree-based hashing scheme is a good fit for GPUs. It also provides means for configuration (e.g., the number of rounds) allowing developers to explore different performance-security trade-offs [6, 7].

7.2 Performance Overheads

Testbed setup. For our evaluation, we used an Intel Xeon E5-1620-v3 server with 8 cores operating at 3.5 GHz, and two different NVIDIA GPUs: GTX 780 with 2304 CUDA cores operating at 863 MHz and GTX Titan Black with 2880 CUDA cores operating at 889 MHz. The general performance trends were similar with both GPUs. Therefore, we present results only for Titan Black. The host CPU runs Linux kernel 4.11 and uses CUDA toolkit v5.0 for GPU kernel compilation.

Command-based API. First, we quantify the overhead of using the command-based API using a matrix-matrix addition microbenchmark. Table 1 shows a breakdown of latency of command execution into five components: **Base**, which is the cumulative latency of all MMIO operations performed during command execution without any security extensions; **Inv.** which is the additional latency of invariant checks including PMM maintenance; **Init** which is the latency for initialization of page directory and page tables; **Crypto** which includes all required cryptographic operations; and interrupt handling, labeled as **Intr**. Note that the measured latency is obtained based on emulation, and therefore overestimates the latency that can be achieved with dedicated hardware support.

We find that the latency of `CH_CREATE` is dominated by the cost of initializing the page directory, and that of `CH_DESTROY` is dominated by the cost of walking the page directory to remove pages of locked page tables from PMM. For `CH_PDE`, we measure the latency for allocating a page directory entry for a small and big page tables. Allocating an entry for a small page table incurs a higher latency because the command needs to reset a larger number of entries. Finally, we measure the latency of `CH_PTE` for allocating an object spanning eight entries of a small page table or a big page table. Here, the latency is higher for big page tables because a larger number of invariant checks using the PMM, which tracks ownership at small page granularity.

Table 1: Command execution latency (μs)

Command	Base	Inv.	Init	Crypto	Intr
`CH_CREATE`	2	13	2246	11	58
`CH_DESTROY`	1	10592	N/A	23	68
`CH_PDE` (S)	1	104	3783	N/A	62
`CH_PDE` (B)	1	62	57	N/A	63
`CH_PTE` (S-8)	8	14	N/A	21	82
`CH_PTE` (B-8)	8	297	N/A	21	78

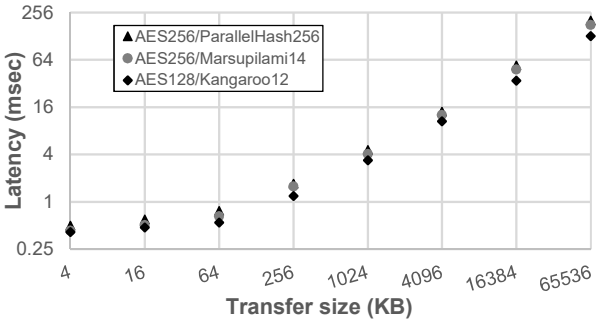


Figure 9: Secure memory copy performance for various sizes and configurations.

Secure memory copy. Figure 9 plots the latency of secure copy for three `AES/SHA3` variants and transfer sizes. The variants `ParallelHash256` and `Marsupilami14` provide 256-bit hashing strength while `Kangaroo12` provides 128-bit hashing strength. We find that latency remains flat for small transfer sizes and scales almost linearly for larger transfer sizes. Unless stated otherwise, we utilize the `AES256/Marsupilami14` configuration for the rest of the evaluation.

Table 2 shows a breakdown of the latency for `AES256/Marsupilami14` configuration. `Base` refers to the latency of normal (insecure) copy, and the other four components refer to the latency of executing AES and SHA-3 on the CPU and GPU. We find that as the transfer size increases, `SHA3-CPU` and `AES-GPU` account for a majority of the overheads (over 75% of the latency for 64MB transfers). For small data sizes, the `AES-GPU` phase, which is compute bound, under-utilizes the GPU cores and hence the execution time remains flat. In contrast, the `SHA3-GPU` kernel scales better due to lower algorithmic complexity. More generally, we attribute both

Table 2: Secure memory copy breakdown for AES256/Marsupilami14. Latency is reported in ms.

Size	4KB	64KB	256KB	4MB	64MB
Base	0.02	0.03	0.08	1.07	11.05
AES-CPU	0.01	0.05	0.10	1.46	25.45
SHA3-CPU	0.02	0.11	0.34	3.19	51.79
SHA3-GPU	0.12	0.13	0.12	0.31	0.58
AES-GPU	0.31	0.38	0.79	6.54	87.98
Total	0.46	0.70	1.43	12.57	176.87

Table 3: CUDA driver API latency (ms)

API	Normal	Secure
cuCtxCreate	77.65	252.63
cuCtxDestroy	17.00	29.43
cuModuleLoad	1.72	85.27
cuMemAlloc (S|B)	0.02|0.03	0.19|0.43
cuMemFree (S|B)	0.03|0.05	0.28|0.66

these costs to the lack of ISA support for SHA-3 on the CPU and for AES on the GPU.

CUDA driver API. Our implementation of secure extensions to the CUDA runtime API are based on extensions to the CUDA driver API. Table 3 shows the impact of adding security on latency of these APIs. As expected, all driver APIs incur higher latencies. The relatively high latency of secure version of context creation `cuCtxCreate` is dominated by the latency of creating an RSA key (75% of latency). The secure version of module load `cuModuleLoad` is more expensive because it (a) bootstraps secure copy, which measures the authenticated encryption kernels and (b) uses secure copy to transfer the application kernels to device memory. These APIs are typically used infrequently, and therefore these latencies do not have a large impact on overall execution time in most applications. On the other hand, `cuMemAlloc` and `cuMemFree` can be on the critical path for applications that use a large number of short-lived objects. The increased latency of these operations is predominantly due to emulation (interrupts and MMIO accesses). We expect an actual implementation of these operations on real Graviton hardware to incur much lower overheads with no involved interrupts and reduced memory access latency.

7.3 Applications

Finally, we quantify the impact of using secure CUDA APIs on end-to-end application performance using a set of GPU benchmarks. We use two benchmarks with different characteristics, namely Caffe, a framework for training and inference of artificial neural networks [18], and BlackScholes, an option pricing application [8].

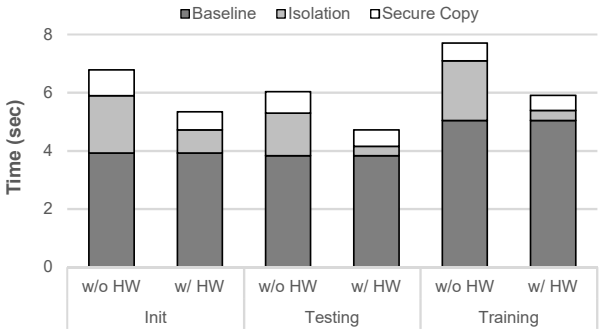


Figure 10: Cifar-10 performance. For training, time is reported for 25 batches averaged across all epochs. HW refers to a hypothetical hardware encryption engine used for command group authenticated decryption.

Cifar-10. We use Caffe to train a convolutional neural network on the Cifar-10 dataset, which consists of 60000 32x32 images spanning 10 classes. The network consists of 11 layers: 3 layers of convolution, pooling, rectified linear unit non-linearities (RELU) followed by local contrast normalization and a linear classifier. We run 40 epochs (each epoch is a sweep over 50000 images) with a batch size of 200 and test the model at the end of every epoch using 10000 images with a batch size of 400. Both the baseline system and Graviton achieve the same training accuracy.

Figure 10 shows Graviton's impact on execution time for three phases of execution—i.e. initialization, testing (which is similar to inference), and training. For training, execution time is reported for 25 batches averaged across all epochs. We also breakdown the overhead into two buckets, isolation (i.e., using the secure CUDA driver API and command group authentication) and secure memory copy. In the emulated environment, our security extensions cause a slowdown of 73%, 57% and 53% respectively in each of these phases.

The overheads during initialization are due to secure context and module creation (22% of the overhead), secure copy of the model and data used for the initial test (31%), and command authentication during an initial testing phase (47%).

A breakdown of testing and training overheads shows that that command group authentication accounts for 66% and 77% of the overhead, respectively. This is because this workload executes a large number of relatively short kernels (one for each batch and layer). We profiled the time spent on kernel launches, and find that a large fraction of the overhead is due to the cost of emulating authenticated decryption of commands. In par-

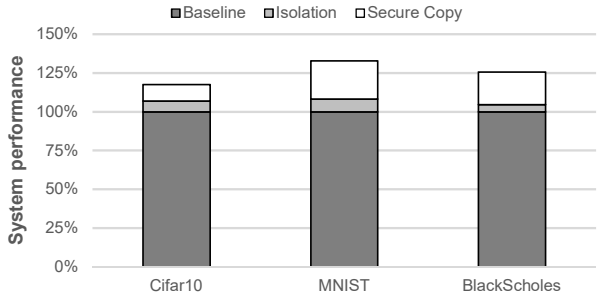


Figure 11: System performance for various benchmarks.

ticular, each secure kernel launch incurs a 9.2μs latency, with 0.8μs on encryption in the runtime, and 3.0μs on decryption in the emulator.

The figure also shows the estimated overhead assuming we extend the command processor with a hardware encryption engine. The overhead reduces from 35-41% to 5-7% for testing and training phases due to a reduction in time spent on authenticated decryption from 3μs to around 30ns. Adding a hardware encryption engine reduces the overall overhead to 17% (Figure 11).

MNIST. We use Caffe to train an autoencoder on the MNIST dataset, which consists of 60000 28x28 handwritten digits. The network consists of six encoding layers and six decoding layers. We run 10000 batches (with a batch size of 128) and test the model every 500 batches using 8192 images with a batch size of 256. Both baseline and Graviton achieve same accuracy.

As shown in Figure 11, Graviton introduces 33% performance overhead. The overhead is higher than in Cifar-10 as the complexity of encoding and decoding layers is lower than convolutional layers, and hence each iteration spends higher fraction time on secure memory copy.

BlackScholes. We run BlackScholes with 10 batches of four million options and 2500 iterations each. As shown in Figure 11, the overall overhead is 26%. Unlike Cifar-10, command authentication is not a factor in BlackScholes as it executes one long-running kernel per batch; thus, the overhead for enforcing isolation is attributed mainly to secure context and module creation.

8 Related Work

Trusted hardware. There is a history of work [9, 13, 15, 17, 24, 27, 32, 41, 48] on trusted hardware that isolates code and data from the rest of the system. Intel SGX [28] is the latest in this line of work, but stands out because it provides comprehensive protection and is already available in client CPUs and public cloud platforms. Graviton effectively extends the trust boundary of TEEs on the CPU to rich devices, such as GPUs.

Trusted execution on GPUs. A number of researchers have identified the need for mechanisms that allow an application hosted in a TEE to securely communicate with I/O devices over a *trusted path*. Yu et al. [51] propose an approach for using the trusted path approach for GPUs. Their approach relies on a privileged host component to enforce isolation between virtual machines and display, whereas our attacker model precludes trust in any host component.

PixelVault proposed an architecture for securely offloading cryptographic operations to a GPU [46]. Subsequent work has demonstrated that such design suffers from security vulnerabilities due to lack of page initialization upon allocations and module creation, lack of kernel-level isolation, and information leakage of registers by either attaching a debugger to a running kernel (and the GPU runtime) or invoking a kernel on the same channel [23, 52]. In contrast, Graviton enables a general-purpose trusted execution environment on GPUs. Information leakage via kernel debugging is prevented as the user hosts its GPU runtime inside a CPU TEE, guaranteeing that debugging cannot be enabled during execution.

9 Conclusion

Unlike recent CPUs, GPUs provide no support for trusted execution environments (TEE), creating a tradeoff between security and performance. In this paper, we introduce Graviton, an architecture for supporting TEEs on GPUs. Our proof-of-concept on NVIDIA GPUs shows that hardware complexity and performance overheads of the proposed architecture are low.

An interesting avenue for future work is to extend Graviton to secure kernel execution and communication across multiple GPUs and to investigate support for advanced features, such as on-demand paging and dynamic thread creation. In addition, we would like to investigate whether it is possible to remove the dependency on CPU TEEs, such as Intel SGX, and if so to quantify the implications on system performance. Finally, we would like to validate whether the proposed architecture extends to other accelerators, such as FPGAs.

Acknowledgements

We would like to thank our shepherd, Andrew Warfield, and the anonymous reviewers for their insightful comments and feedback on our work. Istvan Haller provided input during the early stages of this work. We thank Dushyanth Narayanan and Jay Lorch for comments on earlier drafts of the work.

References

[1] Security on ARM Trustzone. https://www.arm.com/products/security-on-arm/trustzone.

[2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *Interantional Symposium on High Performance Computer Architecture*, 2012.

[3] S. Alhelaly, J. Dworak, T. Manikas, P. Gui, K. Nepal, and A. L. Crouch. Detecting a trojan die in 3D stacked integrated circuits. In *2017 IEEE North Atlantic Test Workshop*, 2017.

[4] AMD. AMD Radeon RX 300 series. https://en.wikipedia.org/wiki/AMD_Radeon_Rx_300_series.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyer, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[6] G. Bertoni, J. Daemen1, M. Peeters, G. V. Assche1, and R. V. Keer. Keccak and the SHA-3 standardization. https://keccak.team/index.html.

[7] G. Bertoni, J. Daemen1, M. Peeters, G. V. Assche1, R. V. Keer, and B. Viguier. KangarooTwelve: Fast hashing based on Keccak-p . In *Cryptology ePrint Archive: Report 2016/770*.

[8] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–654, 1973.

[9] R. Boivie. SecureBlue++: CPU support for secure execution. In *IBM Research Report RC25287*, 2012.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[11] I. Buck. NVIDIAs next-gen Pascal GPU architecture to provide 10X speedup for deep learning apps. https://blogs.nvidia.com/blog/2015/03/17/pascal.

[12] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with DéJà Vu. In *ACM Asia Conference on Computer and Communications Security*, 2017.

[13] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[14] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[15] D. Evtyushkin, J. Elwell, M. Ozsoy, D. V. Ponomarev, N. B. Abu-Ghazaleh, and R. Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *International Symposium on Microarchitecture*, 2014.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31:6–15, 2011.

[17] O. S. Hofmann, S. M. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *International Conference on Multimedia*, 2014.

[19] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, 2017.

[21] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Conference on Annual Technical Conference*, 2012.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[23] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2014.

[24] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[25] B. Madden. NVIDIA, AMD, and Intel: How they do their GPU virtualization. http://www.brianmadden.com/opinion/NVIDIA-AMD-and-Intel-How-they-do-their-GPU-virtualization.

[26] C. Maurice, C. Neumann, and A. Heen, Olivier andFrancillon. Confidentiality issues on a GPU in a virtualized environment. In *Financial Cryptography*, 2014.

[27] J. M. McCune, N. Qu, Y. Li, A. Datta, V. D. Gligor, and A. Perrig. Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2009.

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[29] Nouveau. Accelerated open source driver for NVIDIA cards. `https://nouveau.freedesktop.org/wiki`.

[30] NVIDIA. CUDA multi process service overview. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[31] NVIDIA. Cuda streams. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf`.

[32] E. Owusu, J. Guajardo, J. M. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security*, 2013.

[33] J. J. K. Park, Y. Park, and S. Mahkle. Chimera: Collaborative preemption for multitasking on a shared GPU. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[34] J. J. K. Park, Y. Park, and S. Mahkle. Dynamic resource management for efcient utilization of multitasking GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[35] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[36] R. D. Pietro, F. Lombardi, and A. Villani. CUDA Leaks: A detailed hack for CUDA and a (partial) fix. *ACM Transactions on Embedded Computing Systems*, 15(1), 2016.

[37] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecuture*, 2014.

[38] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[39] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Network and Distributed System Security Symposium*, 2017.

[40] Synopsys. DesignWare pipelined AES-GCM/CTR core. `https://www.synopsys.com/dw/ipdir.php?ds=security-aes-gcm-ctr`.

[41] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[42] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *International Symposium on Computer Architecture*, 2014.

[43] S. Tomov, J. Dongarra, I. Yamazaki, A. Haidar, M. Gates, S. Donfack, P. Luszczek, A. Yarkhan, J. Kurzak, H. Anzt, and T. Dong. MAGMA: Development of high-performance linear algebra for GPUs. In *GPU Technology Conference*, 2014.

[44] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[45] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[46] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for securing cryptographic operations. In *ACM Conference on Computer and Communications Security*, 2014.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *International Conference on High-Performance Computer Architecture*, 2016.

[48] S. Weiser and M. Werner. SGXIO: Generic trusted I/O path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[49] E. Worthman. Designing for security. `https://www.semiengineering.com/designing-for-security-2`.

[50] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, 2015.

[51] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *ACM Conference on Computer and Communications Security*, 2015.

[52] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein. Understanding the security of discrete GPUs. In *Workshop on General Purpose GPUs*, 2017.

ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks

Radhesh Krishnan Konoth[†], Marco Oliverio[†§], Andrei Tatar[†], Dennis Andriesse[†], Herbert Bos[†], Cristiano Giuffrida[†] and Kaveh Razavi[†]

[†] *Vrije Universiteit Amsterdam, The Netherlands*
[§] *Università della Calabria, Italy*

Abstract

The Rowhammer vulnerability common to many modern DRAM chips allows attackers to trigger bit flips in a row of memory cells by accessing the adjacent rows at high frequencies. As a result, they are able to corrupt sensitive data structures (such as page tables, cryptographic keys, object pointers, or even instructions in a program), and circumvent all existing defenses.

This paper introduces ZebRAM, a novel and comprehensive software-level protection against Rowhammer. ZebRAM isolates every DRAM row that contains data with guard rows that absorb any Rowhammer-induced bit flips; the only known method to protect against all forms of Rowhammer. Rather than leaving guard rows unused, ZebRAM improves performance by using the guard rows as efficient, integrity-checked and optionally compressed swap space. ZebRAM requires no hardware modifications and builds on virtualization extensions in commodity processors to transparently control data placement in DRAM. Our evaluation shows that ZebRAM provides strong security guarantees while utilizing all available memory.

1 Introduction

The Rowhammer vulnerability, a defect in DRAM chips that allows attackers to flip bits in memory at locations to which they should not have access, has evolved from a mere curiosity to a serious and very practical attack vector for compromising PCs [6], VMs in clouds [28, 37], and mobile devices [13, 34]. Rowhammer allows attackers to flip bits in DRAM rows simply by repeatedly reading neighboring rows in rapid succession. Existing software-based defenses have proven ineffective against advanced Rowhammer attacks [4, 7], while hardware defenses are impractical to deploy in the billions of devices already in operation [23]. This paper introduces ZebRAM, a comprehensive software-based defense preventing all Rowhammer attacks by isolating every data row in memory with guard rows that absorb any bit flips that may occur.

Practical Rowhammer attacks Rowhammer attacks can target a variety of data structures, from page table entries [30, 34, 36, 37] to cryptographic keys [28], and from object pointers [6, 13, 32] to opcodes [14]. These target data structures may reside in the kernel [30, 34], other virtual machines [28], the same process address space [6, 13], and even on remote systems [32]. The attacks may originate in native code [30], JavaScript [6, 15], or from co-processors such as GPUs [13] and even DMA devices [32]. The objective of the attacker may be to escalate privileges [6, 34], weaken cryptographic keys [28], compromise remote systems [32], or simply lock down the processor in a denial-of-service attack [18].

Today's defenses are ineffective Existing hardware-based Rowhammer defenses fall into three categories: refresh rate boosting, target row refresh, and error correcting codes. Increasing the refresh rate of DRAM [21] makes it harder for attackers to leak sufficient charge from a row before the refresh occurs, but cannot prevent Rowhammer completely without unacceptable performance loss and power consumption increase. The target row refresh (TRR) defense, proposed in the LPDDR4 standard, uses hardware counters to monitor DRAM row accesses and refreshes specific DRAM rows suspected to be Rowhammer victims. However, TRR is not widely deployed; it is optional even in DDR4 [20]. Moreover, researchers still regularly observe bit flips in memory that is equipped with TRR [29]. As for error correcting codes (ECC), the first Rowhammer publication already argued that even ECC-protected DRAM is susceptible to Rowhammer attacks that flip multiple bits per memory word [21]. While this is complicating attacks, they do not stop fully stop them as shown by the recent ECCploit attack [10]. Furthermore, ECC memory is unavailable on most consumer devices.

Software defenses do not suffer from the same deployment issues as hardware defenses. These solutions can be categorized into primitive weakening, detection, and

isolation.

Primitive weakening makes some of the steps in Rowhammer attacks more difficult, for instance by making it harder to obtain physically contiguous uncached memory [30], or to create the cache eviction sets required to access DRAM in case the memory *is* cached. Research has already shown that these solutions do not fundamentally prevent Rowhammer [13].

Rowhammer detection uses heuristics to detect suspected attacks and refresh victim rows before they succumb to bit flips. For instance, ANVIL uses hardware performance counters to identify likely Rowhammer attacks [4]. Unfortunately, hardware performance counters are not available on all CPUs, and some Rowhammer attacks may not trigger unusual cache behavior or may originate from unmonitored devices [13].

A final, and potentially very powerful defense against Rowhammer is to *isolate* the memory of different security domains in memory with unused *guard rows* that absorb bit flips. For instance, CATT places a guard row between kernel and user memory to prevent Rowhammer attacks against the kernel from user space [7]. Unfortunately, CATT does not prevent Rowhammer attacks between user processes, let alone attacks *within* a process that aim to subvert cryptographic keys [28]. Moreover, the lines between security domains are often blurry, even in seemingly clear-cut cases such as the kernel and user-space, where the shared page cache provides ample opportunity to flip bits in sensitive memory areas and launch devastating attacks [14].

ZebRAM: isolate everything from everything Given the difficulty of correctly delineating security domains, the only *guaranteed* approach to prevent all forms of Rowhammer is to isolate *all* data rows with guard rows that absorb bit flips, rendering them harmless. The guard rows, however, break compatibility: buddy allocation schemes (and certain devices) require physically-contiguous memory regions. Furthermore, the drawback of this approach is obvious—sacrificing 50% of memory to guard rows is extremely costly. This paper introduces ZebRAM, a novel, comprehensive and compatible software protection against Rowhammer attacks that isolates everything from everything else *without* sacrificing memory consumed by guard rows. To preserve compatibility, ZebRAM remaps physical memory using existing CPU virtualization extensions. To utilize guard rows, ZebRAM implements an efficient, integrity-checked and optionally compressed swap space in memory.

As we show in Section 7, ZebRAM incurs an overhead of 5% on the SPEC CPU 2006 benchmarks. While ZebRAM remains expensive in the memory-intensive `redis` instance, our evaluation shows that ZebRAM's in-memory swap space significantly improves performance

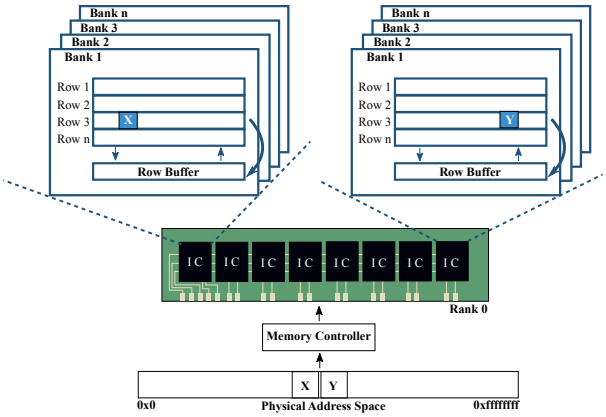


Figure 1: DRAM organization and example mapping of two consecutive addresses.

compared to our basic solution that leaves the guard rows unused, in some cases eliminating over half of the observed performance degradation. In practice, the recent Meltdown/Spectre vulnerabilities show that for a sufficiently serious threat, even expensive fixes are accepted [24]. First and foremost, however, this work investigates an extreme point in the design space of Rowhammer defenses: the first complete protection against all forms of Rowhammer, without sacrificing memory, at a cost that is a function of the workload.

Contributions Our contributions are the followings:

- We describe ZebRAM, the first comprehensive software protection against all forms of Rowhammer.
- We introduce a novel technique to utilize guard rows as fast, memory-based swap space, significantly improving performance compared to solutions that leave guard rows unused.
- We implement ZebRAM and show that it achieves both practical performance and effective security in a variety of benchmark suites and workloads.
- ZebRAM is open source to support future work.

2 Background

This section discusses background on DRAM organization, the Rowhammer bug, and existing defenses.

2.1 DRAM Organization

We now discuss how DRAM chips are organized internally, which is important knowledge for launching an effective Rowhammer attack. Figure 1 illustrates the DRAM organization.

The most basic unit of DRAM storage is a *cell* that can hold a single bit of information. Each DRAM cell consists of two components: a capacitor and a transistor. The capacitor stores a bit by retaining electrical charge. Because this charge leaks away over time, the memory controller periodically (typically every 64 ms) reads each cell and rewrites it, restoring the charge on the capacitor. This process is known as *refreshing*.

DRAM cells are grouped into *rows* that are typically 1024 cells (or *columns*) wide. Memory accesses happen at row granularity. When a row is accessed, the contents of that row are put in a special buffer, called the *row buffer*, and the row is said to be *activated*. After the access, the activated row is written back (i.e., recharged) with the contents of the row buffer.

Multiple rows are stacked together to form *banks*, with multiple banks on a DRAM *integrated circuit (IC)* and a separate row buffer per bank. In turn, DRAM ICs are grouped into *ranks*. DRAM ICs are accessed in parallel; for example, in a DIMM that has eight ICs of 8 bits wide each, all eight ICs are accessed in parallel to form a 64 bit *memory word*.

To address a memory word within a DRAM rank, the system memory controller uses three addresses for the bank, row and column, respectively. Note that the mapping between a physical memory address and the corresponding rank-index, bank-index and row-index on the hardware module is nonlinear. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows (see Figure 1). As explained next, knowledge of the address mapping is vital to effective Rowhammer.

2.2 The Rowhammer Bug

As DRAM chips become denser, the capacitor charge reduces, allowing for increased DRAM capacity and lower energy consumption. Unfortunately, this increases the possibility of memory errors owing to the smaller difference in charge between a "0" bit and a "1" bit.

Research shows that it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring *victim* rows to leak their charge before the memory controller has a chance to refresh them [21]. This rapid activation of memory rows to flip bits in neighboring rows is known as the *Rowhammer attack*. Subsequent research has shown that bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways, including privilege escalation attacks and attacks against co-hosted VMs in cloud environments [6, 15, 27, 28, 30, 34, 37].

The original Rowhammer attack [30] is now known as *single-sided* Rowhammer. As Figure 2 shows, it uses many rapid-fire memory accesses in one *aggressor* row $k-1$ to induce bit flips in a neighboring victim row k. A newer variant called *double-sided* Rowhammer hammers rows $k-1$ and $k+1$ on both sides of the victim row k, increasing the likelihood of a bit flip (see Figure 2). Recent research shows that bit flips can also be induced by hammering only one memory address [14] (*one-location* hammering). Regardless of the type of hammering, Rowhammer can only induce bit flips on directly neighboring DRAM rows.

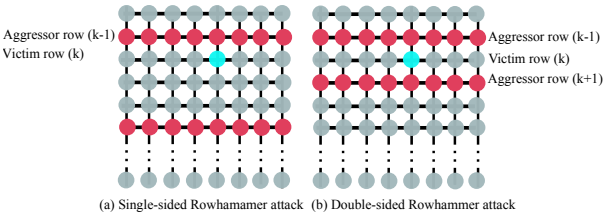


Figure 2: Flipping a bit in a neighboring DRAM row through single-sided (a) and double-sided (b) Rowhammer attacks.

In contrast to single-sided Rowhammer, the double-sided variant requires knowledge of the mapping of virtual and physical addresses to memory rows. Since DRAM manufacturers do not publish this information, this necessitates reverse engineering the DRAM organization.

2.3 Rowhammer Defenses

Research has produced both hardware- and software-based Rowhammer defenses.

The original hardware defense proposed by Kim et al. [21] doubles the refresh rate. Unfortunately, this has been proven insufficient to defend against Rowhammer [4]. Other hardware defenses include error-correcting DRAM chips (ECC memory), which can detect and correct a 1-bit error per ECC word (64-bit data). Unfortunately, ECC memory cannot correct multi-bit errors [3, 23] and is not readily available in consumer hardware. The new LPDDR4 standard [19] specifies two features which together defend against Rowhammer: *Target Row Refresh (TRR)* enables the memory controller to refresh rows adjacent to a certain row, and *Maximum Activation Count (MAC)* specifies a maximum row activation count before adjacent rows are refreshed. Despite these defenses, Gruss et al. [29] still report bit flips in TRR memory.

ANVIL [4], a software defense, uses Intel's performance monitoring unit (PMU) to detect physical addresses that cause many cache misses indicative of Rowhammer.[1] It then recharges suspected victim rows

[1] Rowhammer attacks repeatedly clear hammered rows from the CPU cache to ensure that they hammer DRAM memory, not the cache.

by accessing them. Unfortunately, the PMU does not accurately capture memory accesses through DMA, and not all CPUs feature PMUs. Moreover, the current implementation of ANVIL does not accurately take into account DRAM address mapping and has been reported to be ineffective because of it [31].

Another software-based defense, B-CATT [8], implements a bootloader extension to blacklist all the locations vulnerable to Rowhammer, thus wasting the memory. However, Gruss et al. [14] show that this approach is not practical as it may blacklist over 95% of memory locations; similar results were reported by Tatar et al. [31] showing DIMMs with 99+% vulnerable memory locations. In addition, in our experiments, we have observed different bit flip patterns over time for the same module, making B-CATT incomplete.

Yet another software-based defense called CATT [7] proposes an alternative memory allocator for the Linux kernel that isolates user and kernel space in physical memory, thus ensuring that user-space attackers cannot flip bits in kernel memory. However, CATT does not defend against attacks between user-space processes, and previous work [14] shows that CATT can be bypassed by flipping bits in the code of the `sudo` program.

3 Threat Model

The Rowhammer attacks found in prior research aim for privilege escalation [6, 27, 28, 30, 34, 37, 15], compromising co-hosted virtual machines [28, 37] or even attacks over the network [32]. Our approach, ZebRAM, addresses all these attacks through its principle of isolating memory rows from each other. Our prototype implementation of ZebRAM focuses only on virtual machines, stopping all of the aforementioned attacks launched from or at a victim virtual machine, assuming the hypervisor is trusted. We discuss possible alternative implementations (e.g., native) in Section 9.2.

4 Design

To build a comprehensive solution against Rowhammer attacks, we should consider Rowhammer's fault model: bit flips only happen in adjacent rows when a target row is hammered as shown in Figure 3. Given that any row can potentially be hammered by an attacker, all rows in the system can be abused. To protect against Rowhammer in software, we can follow two approaches: we either need to protect the entire memory against Rowhammer or we need to limit the rows that the attacker can access. Protecting the entire memory is not secure even in hardware [23, 34] and software attempts have so far been shown to be insecure [14]. Instead, we aim to design a

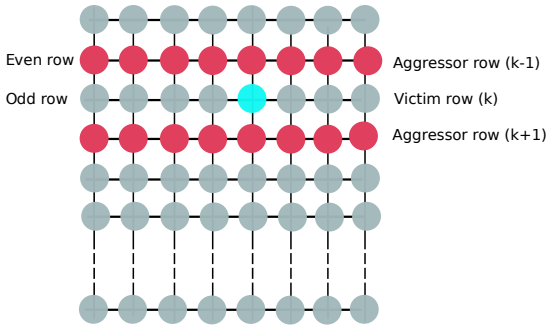


Figure 3: Hammering even-numbered rows can only induce bit flips in odd-numbered rows and vice versa.

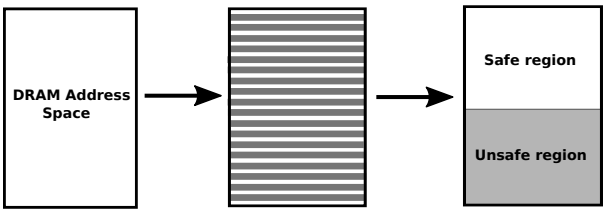


Figure 4: Splitting the memory into safe and unsafe regions using even and odd rows in a zebra pattern.

system where an attacker can only hammer a subset of rows directly.

Basic ZebRAM In order to make sure that Rowhammer bit flips cannot target any data, we should enforce the invariant that all *adjacent rows are unused.* This can be done by making sure that either all odd or all even rows are unused by the system. Assuming odd rows are unused, all even rows will create a *safe region* in memory; it is not possible for an attacker to flip bits in this safe regions simply because all the odd rows are inaccessible to the attacker. The attacker can, however, flip bits in the odd rows by hammering the even rows in the safe region. Hence, we call the odd rows the *unsafe region* in memory. Given that the unsafe region is unused, the attacker cannot flip bits in the data used by the system. This simple design with its zebra pattern shown in Figure 4 already stops all Rowhammer attacks. It however has an obvious downside: it wastes half of the memory that makes up the unsafe region. We address this problem later when we explain our complete ZebRAM design.

A more subtle downside in this design is incompatibility with the Buddy page allocation scheme used in commodity operating systems such as Linux. Buddy allocation requires contiguous regions of physical memory in order to operate efficiently and forcing the system not to use odd rows does not satisfy this requirement. Ideally, our design should utilize the unsafe region while providing (the illusion of) a contiguous physical address

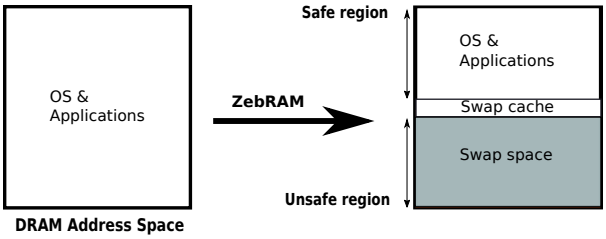


Figure 5: ZebRAM logically divides system memory into a safe region for normal use, a swap space made from the unsafe region, and a swap cache to protect the safe region from accesses made to the unsafe region.

space for efficient buddy allocation as shown on the right side of Figure 4. To address this downside, our design should provide a translation mechanism that creates a linear physical address space out of the safe region.

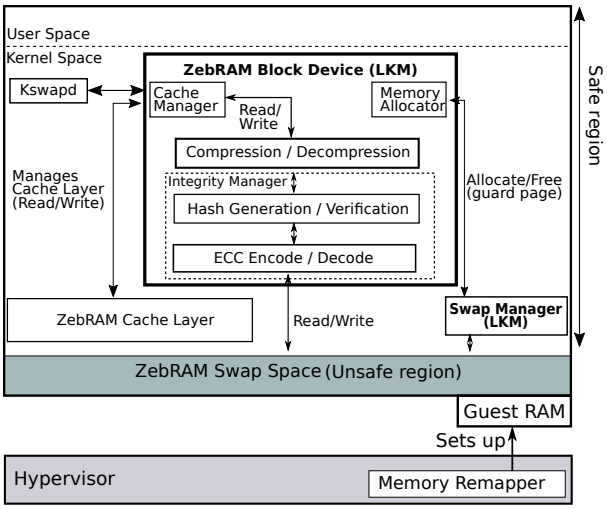


Figure 6: ZebRAM Components.

ZebRAM If we can find a way to securely use the unsafe region, then we can gain back the memory wasted in the basic ZebRAM design. We need to enforce two invariants if we want to make use of the unsafe region for storing data. First, we need to make sure that we properly handle potential bit flips in the unsafe region. Second, we need to ensure that accessing the unsafe region does not trigger bit flips in the safe region. Our proposed design, ZebRAM, shown in Figure 5 satisfies all these requirements. To handle bit flips in the unsafe region, ZebRAM performs software integrity checks and error correction whenever data in the unsafe region is accessed. To protect the safe region from accesses to the unsafe region, ZebRAM uses a cache in front of the unsafe region. This cache is allocated from the safe region and ZebRAM is free to choose its size and replacement policy in a way that protects the safe region. Finally, to provide backward-compatibility with memory management in commodity systems, ZebRAM can employ translation mechanisms provided by hardware (e.g., virtualization extensions in commodity processors) to translate even rows into a contiguous physical address space for the guest.

To maintain good performance, ZebRAM ensures that accesses to the safe region proceed without interposition. As mentioned earlier, this can potentially cause bit flips in the unsafe region. Hence, all accesses to the unsafe region should be interposed for bit flip detection and correction. To this end, ZebRAM exposes the unsafe region as a swap device to the protected operating system. With this design, ZebRAM reuses existing page replacement policies of the operating system to decide which memory pages should be evicted to the swap (i.e., unsafe region). Given that most operating systems use some form of Least Recently Used (LRU), the working set of the system remains in the safe region, preserving performance. Once the system needs to access a page from the unsafe region, the operating system selects a page from the safe region (e.g., based on LRU) and creates necessary meta data for bit flip detection (and/or correction) using the contents of the page and writes it to the unsafe region. At this point, the system can bring the page to the safe region from the unsafe region. But before that, it uses the previously calculated meta data to perform bit flip detection and correction. Note that the swap cache (for protecting the safe region) is essentially part of the safe region and is treated as such by ZebRAM.

Next, we discuss our implementation of ZebRAM's design before analyzing its security guarantees and evaluating its performance.

5 Implementation

In this section, we describe a prototype implementation of ZebRAM on top of the Linux kernel. Our prototype protects virtual machines against Rowhammer attacks and consists of the following four components: the *Memory Remapper*, the *Integrity Manager*, the *Swap Manager*, and the *Cache Manager*, as shown in Figure 6. Our prototype implements Memory Remapper in the hypervisor and the other three components in the guest OS. It is possible to implement all the components in the host to make ZebRAM guest-transparent. We discuss alternative implementations and their associated trade-offs in Section 9.2. We now discuss these components as implemented in our prototype.

5.1 ZebRAM Prototype Components

Memory Remapper implements the split of physical memory into a safe and unsafe region. One region con-

tains all the even-numbered rows, while the other contains all the odd-numbered rows. Note that because hardware vendors do not publish the mapping of physical addresses to DRAM addresses, we need to reverse engineer this mapping following the methodology established in prior work [26, 37, 31].

Because Rowhammer attacks only affect directly neighboring rows, a Rowhammer attack in one region can only incur bit flips in the other region, as shown in Figure 3. In addition, ZebRAM supports the conservative option of increasing the number of guard rows to defend against Rowhammer attacks that target a victim row not directly adjacent to the aggressor row. However, our experience with a large number of vulnerable DRAM modules shows that with the correct translation of memory pages to DRAM locations, bit flips trigger exclusively in rows adjacent to a row that is hammered.

Integrity Manager protects the integrity of the unsafe region. Our software design allows for a flexible choice for error detection and correction. For error correction, we use a commonly-used Single-Error Correction and Double-Error Detection (SECDED) code. As shown in recent work [10], SECDED and other similar BCH codes can still be exploited on DIMMs with large number of bit flips. Our database of Rowhammer bit flips from 14 vulnerable DIMMs [31] shows that only 0.00015% of all memory words with bit flips can bypass our SECDED code (found in 2 of the 14 vulnerable DIMMs) and 0.13% of them can cause a detectable corruption (found in 7 of the 14 vulnerable DIMMs). To provide strong detection guarantees, while providing correction possibilities, ZebRAM provides the possibility to mix SECDED with collision resistant hash functions such as SHA-256 at the cost of extra performance overhead.

Swap Manager uses the unsafe region to implement an efficient swap disk in memory, protected by the Integrity Manager and accessible only by the OS. Using the unsafe region as a swap space has the advantage that the OS will only access the slow, integrity-checked unsafe region when it runs out of fast safe memory. As with any swap disk, the OS uses efficient page replacement techniques to minimize access to it. To maximize utilization of the available memory, the Swap Manager also implements a compression engine that optionally compresses pages stored in the swap space.

Note that ZebRAM also supports configurations with a dedicated swap disk (such as a hard disk or SSD) in addition to the memory-based swap space. In this case, ZebRAM swap is prioritized above any other swap disks to maximize efficiency.

Cache Manager implements a fully associative cache that speeds up access to the swap space while simultaneously preventing Rowhammer attacks against safe rows by reducing the access frequency on memory rows in the unsafe region. The swap cache is faster than the swap disk because it is located in the safe region and does not require integrity checks or compression. Because attackers must clear the swap cache to be able to directly access rows in the unsafe region, the cache prevents attackers from efficiently hammering guard rows to induce bit flips in safe rows.

Because the cache layer sits in front of the swap space, pages swapped out by the OS are first stored in the cache, in uncompressed format. Only if the cache is full does the Cache Manager flush the *least-recently-added (LRA)* entry to the swap disk. The LRA strategy is important, because it ensures that attackers must clear the *entire* cache after every row access in the unsafe region.

5.2 Implementation Details

We implemented ZebRAM in C on an Intel Haswell machine running Ubuntu 16.04 with kernel v4.4 on top a Qemu-KVM v2.11 hypervisor. Next we provide further details on the implementation various components in the ZebRAM prototype.

Memory Remapper To efficiently partition memory into guard rows and safe rows, we use *Second Level Address Translation (SLAT)*, a hardware virtualization extension commonly available in commodity processors. To implement the Memory Remapper component, we patched Qemu-KVM's `mmap` function to expose the unsafe memory rows to the guest machine as a contiguous memory block starting at physical address `0x3ffe0000`. We use a translation library similar to that of Throwhammer [32] for assigning memory pages to odd and even rows in the Memory Remapper component.

Integrity Manager The Integrity Manager and Cache Manager are implemented as part of the ZebRAM block device, and comprise 369 and 192 LoC, respectively. The Integrity Manager uses `SHA-256` algorithm for error detection, implemented in mainline Linux, to hash swap pages, and keeps the hashes in a linear array stored in safe memory. Additionally, the Integrity Manager by default uses an ECC derived from the extended Hamming(63,57) code [16], expurgated to have a message size an integer multiple of bytes. The obtained ECC is a $[64, 56, 4]_2$ block code, providing single error correction and double error detection (SECDED) for each individual (64-bit) memory word—functionally on par with hardware SEC-DED implementations.

Swap Manager The Swap Manager is implemented as a Loadable Kernel Module (LKM) for the guest OS that maintains a stack containing the Page Frame Numbers (PFNs) of free pages in the swap space. It exposes the RAM-based swap disk as a readable and writable block device that we implemented by extending the `zram` compressed RAM block device commonly available in Linux distributions. We changed `zram`'s `zsmalloc` slab memory allocator to only use pages from the Swap Manager's stack of unsafe memory pages. To compress swap pages, we use the LZO algorithm also used by `zram` [1]. The Swap Manager LKM contains 456 LoC while our modifications to `zram` and `zsmalloc` comprise 437 LoC.

Cache Manager The Cache Manager implements the swap cache using a linear array to store cache entries and a radix tree that maps ZebRAM block device page indices to cache entries. By default, ZebRAM uses 2% of the safe region for the swap cache.

Guest Modifications The guest OS is unchanged except for a minor modification that uses Linux's boot memory allocator API (`alloc_bootmem_low_pages`) to reserve the unsafe memory block as swap space at boot time. Our changes to Qemu-KVM comprise 2.6K lines of code (LoC), while the changes to the guest OS comprise only 4 LoC. Furthermore, the Linux kernel may eagerly write dirty pages into the swap device based on its `swappiness` tunable. In ZebRAM, we use a swappiness of 10 instead of the default value of 60 to reduce the number of unnecessary writes to the unsafe region.

6 Security Evaluation

This section evaluates ZebRAM's effectiveness in defending against traditional Rowhammer exploits. Additionally, we show that ZebRAM successfully defends even against more advanced ZebRAM-aware Rowhammer exploits. We evaluated all attacks on a Haswell i7-4790 host machine with 16GB RAM running our ZebRAM-based Qemu-KVM hypervisor on Ubuntu 16.04 64-bit. The hypervisor runs a guest machine with 4GB RAM and Ubuntu 16.04 64-bit with kernel v4.4, containing all necessary ZebRAM patches and LKMs.

6.1 Traditional Rowhammer Exploits

Under ZebRAM's memory model, traditional Rowhammer exploits on system memory only hammer the safe region, and can therefore trigger bit flips only in the integrity-checked unsafe region by construction. We tested the most popular real-world Rowhammer exploit variants to confirm that ZebRAM correctly detects these integrity violations.

In particular, we ran the single-sided Rowhammer exploit published by Google's Project Zero,[2] as well as the one-location[3] and double-sided[4] exploits published by Gruss et al. on our testbed for a period of 24 hours. During this period the single-sided Rowhammer exploit induced two bit flips in the unsafe region, while the one-location and double-sided exploits failed to produce any bit flips. ZebRAM successfully detected and corrected all of the induced bit flips.

The double-sided Rowhammer exploit failed due to ZebRAM's changes in the DRAM geometry, alternating safe rows with unsafe rows. Conventional double-sided exploits attempt to exploit a victim row k by hammering the rows $k-1$ and $k+1$ below and above it, respectively. Under ZebRAM, this fails because the hammered rows are not really adjacent to the victim row, but remapped to be separated from it by unsafe rows. Unaware of ZebRAM, the exploit thinks otherwise based on the information gathered from the Linux' `pagemap`—due to the virtualization-based remapping layer—and essentially behaves like an unoptimized single-sided exploit. Fixing this requires a ZebRAM-aware exploit that hammers two consecutive rows in the safe region to induce a bit flip in the unsafe region. As described next, we developed such an exploit and tested ZebRAM's ability to thwart it.

6.2 ZebRAM-aware Exploits

To further demonstrate the effectiveness of ZebRAM, we developed a ZebRAM-aware double-sided Rowhammer exploit. This section explains how the exploit attempts to hammer both the safe and unsafe regions, showing that ZebRAM detects and corrects all the induced bit flips.

6.2.1 Attacking the Unsafe Region

To induce bit flips in the unsafe region (where the swap space is kept), we modified the double-sided Rowhammer exploit published by Gruss et al. [15] to hammer every pair of two consecutive DRAM rows in the safe region (assuming the attacker is armed with an ideal ZebRAM-aware memory layout oracle) and ran the exploit five times, each time for 6 hours. As Table 1 shows, the first exploit run induced a total of 4,702 bit flips in the swap space, with 4,698 occurrences of a single bit flip in a 64-bit data word and 2 occurrences of a double bit flip in a 64-bit word. ZebRAM successfully corrected all 4,698 single bit flips and detected the double bit flips. As shown

[2] `https://github.com/google/rowhammer-test`

[3] `https://github.com/IAIK/flipfloyd`

[4] `https://github.com/IAIK/rowhammerjs/tree/master/native`

Run no.	1 bit flip in 64 bits	2 bit flips in 64 bits	Total bit flips	ZebRAM detection performance	
				Detected bit flips	Corrected bit flips
1	4,698	2	4,702	4,702	4,698
2	5,132	0	5,132	5,132	5,132
3	2,790	0	2,790	2,790	2,790
4	4,216	1	4,218	4,218	4,216
5	3,554	0	3,554	3,554	3,554

Table 1: ZebRAM's effectiveness defending against a ZebRAM-aware Rowhammer exploit.

in Table 1, the other exploit runs produced similar results, with no bit flips going undetected. Note that ZebRAM can also detect more than two errors per 64-bit word due to its combined use of ECC and hashing, although our experiments produced no such cases.

6.2.2 Attacking the Safe Region

In addition to hammering safe rows, attackers may also attempt to hammer unsafe rows to induce bit flips in the safe region. To achieve this, an attacker must trigger rapid writes or reads of pages in the swap space. We modified the double-sided Rowhammer exploit to attempt this by opening the swap space with the *open* system call with the *O_DIRECT* flag, followed by repeated *preadv* system calls to directly read from the ZebRAM swap disk (bypassing the Linux page cache).

Because the swap disk only supports page-granular reads, the exploit must read an entire page on each access. Reading a ZebRAM swap page results in at least two memory copies; first to the kernel block I/O buffer, and next to user space. The exploit evicts the ZebRAM swap cache before each swap disk read to ensure that it accesses rows in the swap disk rather than in the cache (which is in the safe region). After each page read, we use a `clflush` instruction to evict the cacheline we use for hammering purposes. Note that this makes the exploit's job easier than it would be in a real attack scenario, where the exploit cannot use `clflush` because the attacker does not own the swap memory. A real attack would require walking an entire cache eviction buffer after each read from the swap disk.

We ran the exploit for 24 hours, during which time the exploit failed to trigger any bit flips. This demonstrates that the slow access frequency of the swap space—due to its page granularity, integrity checking, and the swap cache layer—successfully prevents Rowhammer attacks against the safe region.

To further verify the reliability of our approach, we re-tested our exploit with the swap disk's cache layer, compression engine, and integrity checking modules disabled, thus providing overly optimistic access speeds (and security guarantees) to the swap space for the Rowhammer exploit. Even in this scenario, the page-granular read enforcement of the swap device alone proved sufficient to prevent any bit flips. Our time measurements using `rdtsc` show that even in this optimistic scenario, memory dereferences in the swap space take 2,435 CPU cycles, as opposed to 200 CPU cycles in the safe region, removing any possibility of a successful Rowhammer attack against the safe region.

7 Performance Evaluation

This section measures ZebRAM's performance in different configurations compared to an unprotected system under varying workloads. We test several different kinds of applications, commonly considered for evaluation by existing systems security defenses. First, we test ZebRAM on the SPEC CPU2006 benchmark suite [17] to measure its performance for CPU-intensive applications. We also benchmark ZebRAM the popular `nginx` and Apache web servers, as well as the `redis` in-memory key-value store. Additionally, we present microbenchmark results to better understand the contributing factors to ZebRAM's overhead.

Testbed Similar to our security evaluation, we conduct our performance evaluation on a Haswell i7-4790 machine with 16GB RAM running Ubuntu 16.04 64-bit with our modified Qemu-KVM hypervisor. We run the ZebRAM modules and the benchmarked applications in an Ubuntu 16.04 guest VM with kernel v4.4 and 4GB of memory using a split of 2GB for the safe region and 2GB for the unsafe region. To establish a baseline, we use the same guest VM with an unmodified kernel and 4GB of memory. In the baseline measurements, the guest VM has direct access to all its memory, while in the ZebRAM performance measurements half of the memory is dedicated to the ZebRAM swap space. In all reported memory usage figures we include memory used by the Integrity Manager and Cache Manager components of ZebRAM. For our tests of server applications, we use a separate Skylake i7-6700K machine as the client. This machine has 16GB RAM and is linked to the ZebRAM machine via a 100Gbit/s link. We repeat all our experiments multiple times and observe marginal deviations across runs.

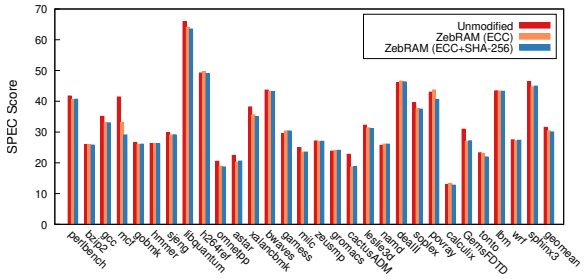


Figure 7: SPEC CPU 2006 performance results.

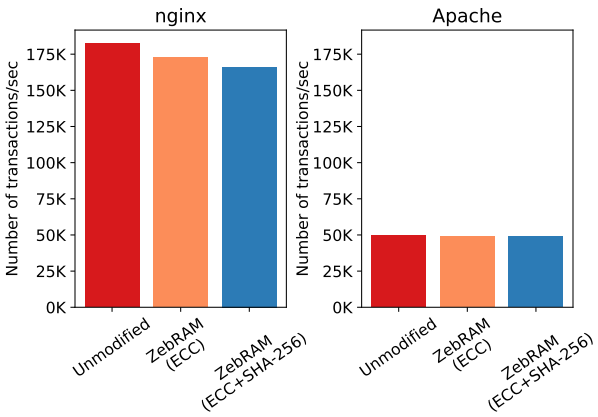


Figure 8: Nginx and Apache throughput at saturation.

SPEC 2006 We compare performance scores of the SPEC 2006 benchmark suite in three different setups: (i) unmodified, (ii) ZebRAM configured to use only ECC, and (iii) ZebRAM configured to use ECC and SHA-256. The ZebRAM (ECC) and ZebRAM (ECC and SHA-256) show a performance overhead over the unmodified baseline of 4% and 5%, respectively (see Figure 7). The reason behind this performance overhead is that as the ZebRAM splits the memory in a zebra pattern, the OS can no longer benefit from huge pages. Also, note that certain benchmarks, such as `mcf`, exhibits more than 5% overhead because they use ZebRAM's swap memory as their working set do not fit in the safe region.

Web servers We evaluate two popular web servers: nginx (1.10.3) and Apache (2.4.18). We configure the virtual machine to use 4 VCPUs. To generate load to the web servers we use the wrk2 [2] benchmarking tool, retrieving a default static HTML page of 240 characters. We set up nginx to use 4 workers, while we set up Apache with the prefork module, spawning a new worker process for every new connection. We also increase the maximum number of clients allowed by Apache from 150 to 500. We configured the wrk2 tool to use 32 parallel keep-alive connections across 8 threads. When measuring the throughput we check that CPU is saturated in the server VM. We discard the results of 3 warmup rounds, repeat a one-minute run 11 times, and report the median across runs. Figure 8 shows the throughput of ZebRAM under two different configurations: (i) ZebRAM configured to use only ECC, and (ii) ZebRAM configured to use ECC and SHA-256. Besides throughput, we want to measure ZebRAM's latency impact. We use wrk2 to throttle the load on the server (using the rate parameter) and report the 99th percentile latency as a function of the client request rate in Figure 9.

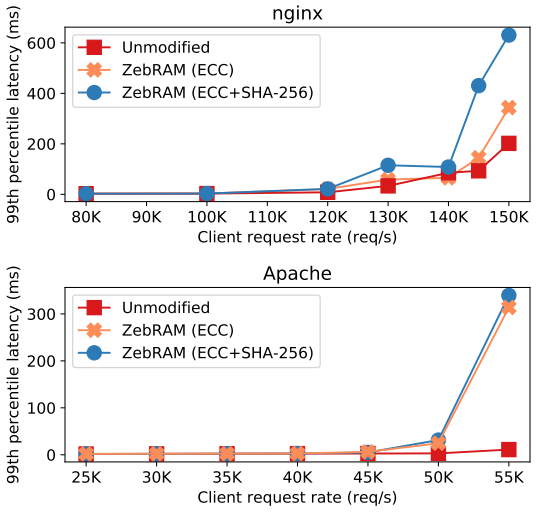


Figure 9: Nginx and Apache latency (99th percentile).

The baseline achieves 182k and 50k requests per second on Nginx and Apache respectively. The ZebRAM's first configuration (only ECC) reaches 172k and 49k while the second configuration reaches 166k and 49k.

Before saturation, the results show that ZebRAM imposes no overhead on the 99th percentile latency. After then, both configurations of ZebRAM show a similar trend with linearly higher 99th percentile response time.

Overall, ZebRAM's performance impact on both web servers and SPEC benchmarks is low and mostly due to the inability to efficiently use Linux' THP support. This is expected, since as long as the working set can comfortably fit in the safe region (e.g., around 400MB for our web server experiments) the unsafe memory management overhead is completely masked. We isolate and study such overhead in more detail in the following.

Microbenchmarks To drill down the overhead of each single feature of ZebRAM, we measure the latency of

swapping in a page from the ZebRAM device under different configurations. To measure the latency, we use a small binary that sequentially writes on every page of a large eviction buffer in a loop. This ensures that, between two accesses to the same page, we touch the entire buffer, evicting that page from memory. To be sure that Linux swaps in just one page for every access, we set the page-cluster configuration parameter to 0. In this experiment, two components interact with ZebRAM: our binary triggers swap-in events from the ZebRAM device while the `kswapd` kernel thread swaps pages to the ZebRAM device to free memory. The interaction between them is completely different if the binary uses exclusively loads to stress the memory. This is because the kernel would optimize out unnecessary flushes to swap and batch together TLB invalidations. Hence, we choose to focus on stores to study the performance in the worst-case scenario and because read-only workloads are less common than mixed workloads.

We reserve a core exclusively for the binary so that `kswapd` does not (directly) steal CPU cycles from it. We measure 1,000,000 accesses for each different configuration. Table 2 presents our results.We also run the binary in a loop and profile its execution with the `perf` Linux tool to measure the time spent in different functions. Due to function inlining, it is not always trivial to map a symbol to a particular feature. Nevertheless, `perf` can provide insights into the overhead at a fine granularity. In the first configuration, we disable the all features of ZebRAM and perform only memory copies into the ZebRAM device. As the copy operation is fast, the `perf` tool reports that just 4% percent of CPU cycles are spent copying. Interestingly, 47% of CPU cycles are spent serving Inter Process Interrupts from other cores. This is because, while we are swapping in, `kswapd` on another core is busy freeing memory. For this purpose, `kswapd` needs to unmap pages that are on their way to be swapped out from the process's page tables. This introduces TLB shootdowns (and IPIs) to invalidate other cores' TLB stale entries. It is important to notice that the faster we swap in pages, the faster `kswapd` needs to free memory. This unfortunately results in a negative feedback loop that represent one of the major sources of overhead when the large number of swap-in events continuously force `kswapd` to wake up.

Adding hashing (SHA-256) on top of the previous configuration shows an increase in latency, which is also reflected in the CPU cycles breakdown. The `perf` tool reports that 55% of CPU cycles are spent swapping in pages (copy + hashing), while serving IPIs accounts for 29%. Adding cache and compression on top of SHA-256 decreases the latency median and increases the 99th percentile. This is because, on a cache hit, the ZebRAM only needs to copy the page to userspace; however, on a cache miss, it has to verify the hash of the page and decompress the page too. The `perf` tool reports 42% of CPU cycles are spent in the decompression routine and 26% in serving IPI requests for other cores and less than 5% in hashing and copying. This confirms the presence of the swap-in/swap-out feedback loop under high memory pressure. Adding ECC marginally increases the latency, the `perf` tool reports similar CPU usage breakdown for the version without ECC.

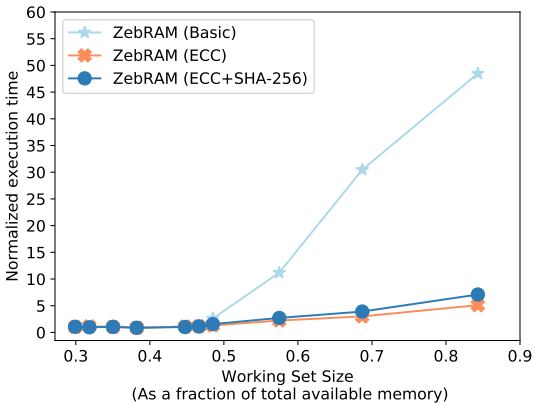


Figure 10: Redis throughput at saturation.

Larger working sets As expected, ZebRAM's overheads are mostly associated to swap-in/swap-out operations, which are masked when the working set can fit in the safe region but naturally become more prominent as we grow the working set. In this section, we want to evaluate the impact of supporting increasingly larger working sets compared to a more traditional swap implementation. For this purpose, we evaluate the performance of a key-value store in four different setups: (i) unmodifed system, (ii) the basic version of ZebRAM (iii) ZebRAM configured with ECC, and (iv) ZebRAM configured with ECC and SHA-256. The basic version of ZebRAM uses just one of the two domains in which ZebRAM splits the RAM and swaps to a fast SSD disk when the memory used by the OS does not fit into it. We use YCSB[11] to generate load and induce a target working set size against a redis (4.0.8) key-value store. We setup YCSB to use 1KB objects and perform a 90/10 read/write operations ratio. Each test runs for 20 seconds and, for each configuration, we discard the results of 3 warmup rounds and report the median across 11 runs. We configure YCSB to access the dataset key space uniformly and we measure the throughput at saturation for different data set sizes.

Figure 10 depicts the reported normalized execution time as a function of the working set size (in percentage compared to the total RAM size). As shown in the figure, when the working set size is small enough (e.g.,

Configuration	median (ns)	90th (ns)	99th (ns)
copy	2,362.0	4,107.0	8,167.0
SHA-256	13,552.0	14,209.0	17,092.0
cache + comp + SHA-256	8,633.0	13,191.0	18,678.0
cache + comp + SHA-256 + ECC	9,862.0	15,118.0	20,794.0

Table 2: Page swap-in latency from the ZebRAM device.

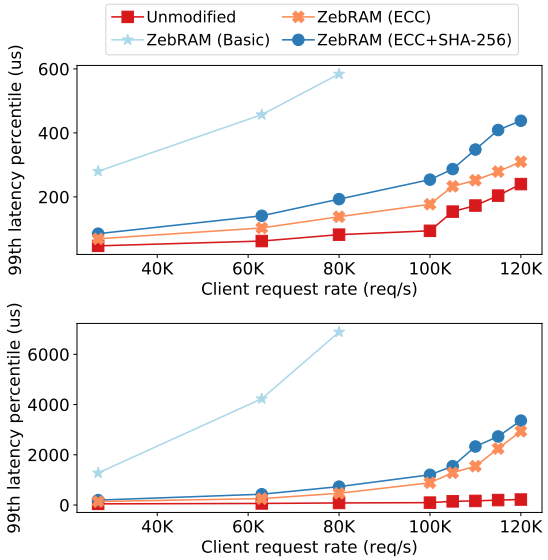


Figure 11: Redis latency (99th percentile). The working set size is 50% of RAM (top) and 70% of RAM (bottom).

44%) the OS hardly reclaims any memory, hence the unsafe region remains unutilized and the normalized execution time is only 1.08x for the basic version of ZebRAM while the normalized execution time is between 1.04x and 1.10x for all other configurations of ZebRAM. As we increase the working set size, the OS starts reclaiming pages and the normalized execution time increases accordingly. However, the increase is much more gentle for ZebRAM compared to the basic version of ZebRAM and the gap becomes more significant for larger working set sizes. For instance, for a fairly large working set size (e.g., 70%), ZebRAM (ECC) has 3.00x normalized execution time, and ZebRAM (ECC and SHA-256) has 3.90x, compared to the basic version of ZebRAM at 30.47x.

To study the impact of ZebRAM on latency, we fix the working set size to 50% and 70% of the total RAM and repeat the same experiment while varying the load on the server. Figure 11 presents our results for the 99th latency percentile. At 50%, results of (i) the ZebRAM configured with ECC, (ii) the ZebRAM configured with ECC and SHA-256, and (iii) baseline (unmodified) follow the same trend. The ZebRAM's first configuration (only ECC) reports a 99th latency percentile of 138us for client request rates below 80,000, compared to 584us for ZebRAM (basic). At 70%, the gap is again more prominent, with ZebRAM reporting a 99th latency percentile of 466us and ZebRAM (basic) reporting 6,887us.

Overall, ZebRAM can more gracefully reduce performance for larger working sets compared to a traditional (basic ZebRAM) swap implementation, thanks to its ability to use an in-memory cache and despite the integrity checks required to mitigate Rowhammer. As our experiments demonstrate, given a target performance budget, ZebRAM can support much larger working sets compared to the ZebRAM's basic implementation, while providing a strong defense against arbitrary Rowhammer attacks. This is unlike the basic ZebRAM implementation, which optimistically provides no protection against similar bit flip-based attacks. Unfortunately, such attacks, which have been long-known for DRAM [21], have recently started to target flash memory as well [9, 22].

8 Related work

This section summarizes related work on Rowhammer attacks and defenses.

Attacks In 2014, Kim et al. [21] were the first to show that it is possible to flip bits in DDR3 memory on x86 CPUs simply by accessing other parts of memory. Since then, many studies have demonstrated the effectiveness of Rowhammer as a real-world exploit in many systems.

The first practical Rowhammer-based privilege escalation attack, by Seaborn and Dullien [30], targeted the x86 architecture and DDR3 memory, hammering the memory rows by means of the native x86 `clflush` instruction that would flush the cache and allow high-frequency access to DRAM. By flipping bits in page table entries, the attack obtained access to privileged pages.

Not long after these earliest attacks, researchers greatly increased the threat of Rowhammer attacks by showing that is possible to launch them from JavaScript also, allowing attackers to gain arbitrary read/write access to the browser address space from a malicious web page [6, 15].

Moreover, newer attacks started flipping bits in memory areas other than page table entries, such as object pointers (to craft counterfeit objects [6]), opcodes [14], and even application-level sensitive data [28].

For instance, Flip Feng Shui demonstrated a new attack on VMs in cloud environments that flipped bits in RSA keys in victim VMs to make them easy to factorize, by massaging the physical memory of the co-located VMs to land the keys on a page that was hammerable by the attacker. Around the same time, other researchers independently also targeted RSA keys with Rowhammer but now for fault analysis [5]. Concurrently, also, Xiao et al. [37] presented another cross-VM attack that manipulates page table entries in Xen.

Where the attacks initially focused on PCs with DDR3 configurations, later research showed that ARM processors and DDR4 memory chips are also vulnerable [34]. While this opened the way for Rowhammer attacks on smartphones, the threat was narrower than on PCs, as the authors were not yet able to launch such attacks from JavaScript. This changed recently, when research described a new way to launch Rowhammer attacks from JavaScript on mobile phones and PC, by making use of the GPU. Hammering directly from the GPU by way of WebGL, the authors managed to compromise a modern smart phone browser in under two minutes. Moreover, this time the targeted data structures are doubles and pointers: by flipping a bit in the most significant bytes, the attack can turn pointers into doubles (making them readable) and doubles into pointers (yielding arbitrary read/write access).

All Rowhammer attacks until that point required local code execution. Recently, however, researchers demonstrated that even remote attacks on servers are possible [32], by sending network traffic over high-speed network to a victim process, using RDMA NICs. As the server that is receiving the network packets is using DMA to write to its memory, the remote attacker is able to flip bits in the server. By carefully manipulating the data in a key-value store, they show that it is possible to completely compromise the server process.

It should be clear that Rowhammer exploits have spread from a narrow and arcane threat to target two of the most popular architectures, in all common computing environments, different types of memory (and arguably flash [9]), while covering most common threat models (local privilege escalation, hosted JavaScript, and even remote attacks). ZebRAM protects against all of the above attacks.

Defenses Kim et al. [21] propose hardware changes to mitigate Rowhammer by increasing row refresh rates or using ECC. These defenses have proven insufficient [4] and infeasible to deploy on the required massive scale. The new LPDDR4 standard [19] specifies two features which together defend against Rowhammer: TRR and MAC. Despite these defenses, van der Veen et al. still report bit flips on a Google pixel phone with LPDDR4 memory [35] and Gruss et al. [29] report bit flips in TRR memory. While nobody has demonstrated Rowhammer attacks against ECC memory yet, the real problem with such hardware solutions is that most systems in use today do not have ECC, and replacing all DRAM in current devices is simply infeasible.

In order to protect from Rowhammer attacks, many vendors simply disabled features in their products to make life harder for attackers. For instance, Linux disabled unprivileged access to the *pagemap* [30], Microsoft disabled memory deduplication [12] to defend from the Dedup Est Machina attack [6], and Google disabled [33] the ION contiguous heap in response to the Drammer attack [34] on mobile ARM devices. Worryingly, not a single defence is currently deployed to protect from the recent GPU-based Rowhammer attack on mobile ARM devices (and PCs), even though it offers attackers a huge number of vulnerable devices.

Finally, researchers have proposed targeted software-based solutions against Rowhammer. ANVIL [4] relies on Intel's performance monitoring unit (PMU) to detect and refresh likely Rowhammer victim rows. An improved version of ANVIL requires specialized Intel PMUs with a fine-grained physical to DRAM address translation. Unfortunately, Intel's (and AMD's) PMUs do not capture precise address information when memory accesses bypass the CPU cache through DMA. Hence, this version of ANVIL is vulnerable to off-CPU Rowhammer attacks. Unlike ANVIL, ZebRAM is secure against off-CPU attacks, since device drivers transparently allocate memory from the safe region.

CATT [7] isolates (only) user and kernel space in physical memory so that user-space attackers cannot trigger bit flips in kernel memory. However, research [14] shows CATT to be bypassable by flipping opcode bits in the `sudo` program code. Moreover, CATT does not defend against attacks that target co-hosted VMs at all [7]. In contrast, ZebRAM protects against co-hosted VM attacks, attacks against the kernel, attacks between (and even within) user-space processes and attacks from co-processors such as GPUs.

Other recent software-based solutions have targeted specific Rowhammer attack variants. GuardION isolates DMA buffers to protect mobile devices against DMA-based Rowhammer attacks [36]. ALIS isolates RDMA buffers to protect RDMA-enabled systems against Throwhammer [32]. Finally, VUSion randomizes page frame allocation to protect memory deduplication-enabled systems against Flip Feng Shui [25].

9 Discussion

This section discusses feature and performance tradeoffs between our ZebRAM prototype and alternative ZebRAM implementations.

9.1 Prototype

Because the ZebRAM prototype relies on the hypervisor to implement safe/unsafe memory separation, and on a cooperating guest kernel for swap management, both host and guest need modifications. In addition, the guest physical address space will map highly non-contiguously to the host address space, preventing the use of huge pages. The guest modifications, however, are small and self-contained, do not touch the core memory management implementation and are therefore highly compatible with mainline and third party LKMs.

9.2 Alternative Implementations

In addition to our implementation presented in Section 5, several alternative ZebRAM implementations are possible. Here, we compare our ZebRAM implementation to alternative hardware-based, OS-based, and guest-transparent virtualization-based implementations.

Hardware-based Implementing ZebRAM at the hardware level would require a physical-to-DRAM address mapping where sets of odd and even rows are mapped to convenient physical address ranges, for instance an even lower-half and an odd upper-half. This can be achieved with by a fully programmable memory controller, or implemented as a configurable feature in existing designs. With such a mapping in place, the OS can trivially separate memory into safe and unsafe regions. In this model, the Swap Manager, Cache Manager and Integrity Manager are implemented as LKMs just as in the implementation from Section 5. In contrast to other implementations, a hardware implementation requires no hypervisor, allows the OS to make use of (transparent) huge pages and requires minimal modifications to the memory management subsystem. While a hardware-supported ZebRAM implementation has obvious performance benefits, it is currently infeasible to implement because memory controllers lack the required features.

OS-based Our current ZebRAM prototype implements the Memory Remapper as part of a hypervisor. Alternatively, the Memory Remapper can be implemented as part of the bootloader, using Linux' boot memory allocator to reserve the unsafe region for use as swap space. While this solution obviates the use of a hypervisor, it also results in a non-contiguous physical address space that precludes the use of huge pages and breaks DMA in older devices. In addition, it is likely that this approach requires invasive changes to the memory management subsystem due to the very fragmented physical address space.

Transparent Virtualization-based While our current ZebRAM implementation requires minor changes to the guest OS, it is also possible to implement a virtualization-based variant of ZebRAM that is completely transparent to the guest. This entails implementing the ZebRAM swap disk device in the host and then exposing the disk to the guest OS as a normal block device to which it can swap out. The drawback of this approach is that it degrades performance by having the hypervisor interposed between the guest OS and unsafe memory for each access to the swap device, a problem which does not occur in our current implementation. The clear advantage to this approach is that it is completely guest-agnostic: guest kernels other than Linux, including legacy and proprietary ones are equally well protected, enabling existing VM deployments to be near-seamlessly transitioned over to a Rowhammer-safe environment.

10 Conclusion

We have introduced ZebRAM, the first comprehensive software defense against all forms of Rowhammer. ZebRAM uses guard rows to isolate all memory rows containing user or kernel data, protecting these from Rowhammer-induced bit flips. Moreover, ZebRAM implements an efficient integrity-checked memory-based swap disk to utilize the memory sacrificed to the guard rows. Our evaluation shows ZebRAM to be a strong defense able to use all available memory at a cost that is a function of the workload. To aid future work, we release ZebRAM as open source.

Acknowledgements

We would like to thank our shepherd, Xi Wang, and the anonymous reviewers for their valuable feedback. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and the UNICORE project, by the MALPAY project, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", NWO 016.Veni.192.262, and NWO 629.002.204 "Parallax". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] LZO. http://www.oberhumer.com/opensource/lzo/, Retrieved 09.09.2018.

[2] WRK2 - a HTTP Benchmarking Tool. https://github.com/giltene/wrk2, Retrieved 09.09.2018.

[3] AICHINGER, B. DDR Memory Errors caused by Row Hammer. HPEC'15.

[4] AWEKE, Z. B., YITBAREK, S. F., QIAO, R., DAS, R., HICKS, M., OREN, Y., AND AUSTIN, T. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. ASPLOS'16.

[5] BHATTACHARYA, S., AND MUKHOPADHYAY, D. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. CHESS'16.

[6] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. S&P'16.

[7] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., AND SADEGHI, A.-R. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. SEC'17.

[8] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., AND SADEGHI, A.-R. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. *arXiv preprint arXiv:1611.08396* (2016).

[9] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O., AND HARATSCH, E. F. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. HPCA '17.

[10] COJOCAR, L., RAZAVI, K., GIUFFRIDA, C., AND BOS, H. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. S&P '19.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. SoCC'10.

[12] CVE-2016-3272. Microsoft Security Bulletin MS16-092 - Important. *https://technet.microsoft.com/en-us/library/security/ms16-092.aspx* (2016).

[13] FRIGO, P., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. S&P'18.

[14] GRUSS, D., LIPP, M., SCHWARZ, M., GENKIN, D., JUFFINGER, J., OCONNELL, S., SCHOECHL, W., , AND YAROM, Y. Another Flip in the Wall of Rowhammer Defenses. *arXiv preprint arXiv:1710.00551* (2017).

[15] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16.

[16] HAMMING, R. W. Error detecting and error correcting codes. *Bell Labs Technical Journal 29*, 2 (1950), 147–160.

[17] HENNING, J. L. SPEC CPU2006 memory footprint. ACM SIGARCH Computer Architecture'07.

[18] JANG, Y., LEE, J., LEE, S., AND KIM, T. Sgx-bomb: Locking down the processor via rowhammer attack. SysTEX'17.

[19] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. Low Power Double Data 4 (LPDDR4). *JESD209-4A* (2015).

[20] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. DDR4 SDRAM Specification. *JESD79-4B* (2017).

[21] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14.

[22] KURMUS, A., IOANNOU, N., PAPANDREOU, N., AND PARNELL, T. From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks. WOOT'17.

[23] LANTEIGNE, M. *How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware* (2016).

[24] NEWMAN, L. H. The hidden toll of fixing meltdown and spectre. *WIRED* (2018).

[25] OLIVERIO, M., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Secure page fusion with vusion. SOSP'17.

[26] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16.

[27] QIAO, R., AND SEABORN, M. A New Approach for Rowhammer Attacks. HOST'16.

[28] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16.

[29] SCHWARZ, M., GRUSS, D., AND LIPP, M. Another Flip in the Row. BHUS'18. https://gruss.cc/files/us-18-Gruss-Another-Flip-In-The-Row.pdf Retrieved 09.09.2018.

[30] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. BHUS'15.

[31] TATAR, A., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Defeating software mitigations against Rowhammer: A surgical precision hammer. RAID'18.

[32] TATAR, A., KRISHNAN, R., ATHANASOPOULOS, E., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Throwhammer: Rowhammer Attacks over the Network and Defenses. ATC'18.

[33] TJIN, P. android-7.1.0_r7 (Disable ION_HEAP_TYPE_SYSTEM_CONTIG). *https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0_r7* (2016).

[34] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. CCS'16.

[35] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. http://vvdveen.com/publications/drammer.slides.pdf, Retrieved 09.09.2018.

[36] VAN DER VEEN, V., LINDORFER, M., FRATANTONIO, Y., PILLAI, H. P., VIGNA, G., KRUEGEL, C., BOS, H., AND RAZAVI, K. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. DIMVA'18.

[37] XIAO, Y., ZHANG, X., ZHANG, Y., AND TEODORESCU, R. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. SEC'16.

Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis

David Lazar, Yossi Gilad, and Nickolai Zeldovich
MIT CSAIL

Abstract

Karaoke is a system for low-latency metadata-private communication. Karaoke provides differential privacy guarantees, and scales better with the number of users than prior such systems (Vuvuzela and Stadium). Karaoke achieves high performance by addressing two challenges faced by prior systems. The first is that differential privacy requires continuously adding noise messages, which leads to high overheads. Karaoke avoids this using *optimistic indistinguishability*: in the common case, Karaoke reveals no information to the adversary, and Karaoke clients can detect precisely when information may be revealed (thus requiring less noise). The second challenge lies in generating sufficient noise in a distributed system where some nodes may be malicious. Prior work either required each server to generate enough noise on its own, or used expensive verifiable shuffles to prevent any message loss. Karaoke achieves high performance using *efficient noise verification*, generating noise across many servers and using Bloom filters to efficiently check if any noise messages have been discarded. These techniques allow our prototype of Karaoke to achieve a latency of 6.8 seconds for 2M users. Overall, Karaoke's latency is 5× to 10× better than Vuvuzela and Stadium.

1 Introduction

Text messaging systems are often vulnerable to traffic analysis, which reveals communication patterns like who is communicating with whom. Hiding this information can be important for some users, such as journalists and whistleblowers. However, building a messaging system just for whistleblowers is not a good idea, because using this system would be a clear indication of who is a whistleblower [9]. Thus, it is important to build metadata-private messaging systems that can support a large number of users with acceptable performance, so as to provide "cover" for sensitive use cases.

A significant limitation of prior work, such as Vuvuzela [26], Pung [1], and Stadium [25], is that they incur high latency. For example, with 2 million connected users, Vuvuzela has an end-to-end latency of 55 seconds, and the latencies of Pung and Stadium are even higher. Such high latencies hinder the adoption of these designs.

This paper presents Karaoke, a metadata-private messaging system that reduces latency by an order of magnitude compared to prior work. For instance, Karaoke achieves an end-to-end latency of 6.8 seconds for 2 million connected users on 100 servers (on Amazon EC2 with simulated 100 msec round-trip latency between servers), 80% of which are assumed to be honest, and achieves differential privacy guarantees comparable to Vuvuzela and Stadium. Furthermore, Karaoke can maintain low latency even as the number of users grows, by scaling horizontally (i.e., having independent organizations contribute more servers). Karaoke supports 16 million users with 28 seconds of latency, a 10× improvement over Stadium.

Achieving high performance requires Karaoke to address two challenges. The first challenge is that differential privacy typically requires adding noise to limit data leakage. Prior work achieves differential privacy for private messaging by enumerating what metadata an adversary could observe (e.g., the number of messages exchanged in a round of communication), and adding fake messages ("noise") that are mixed with real messages to obscure this information. This translates into a large number of noise messages that have to be added every round, and handling these noise messages incurs a high performance cost.

Karaoke addresses this challenge using *optimistic indistinguishability*. Karaoke's design avoids leaking information in the common case, when there are no active attacks. Karaoke further ensures that clients can precisely detect whether any information was leaked (e.g., due to an active attack), so that the clients can stop communicating to avoid leaking more data. This allows Karaoke to add fewer noise messages, because the noise messages need to mask fewer message exchanges (namely, just those where an active attack has occurred).

The second challenge lies in generating the noise in the presence of malicious servers. One approach is to require every server to generate all of the noise on its own, under the assumption that every other server is malicious [26]. This scheme leads to an overwhelming number of noise messages as the number of servers grows. Another approach is to distribute noise generation across many servers. However, a malicious server might drop the noise messages before they are mixed with messages from legitimate users. As a result, achieving privacy requires the use of expensive zero-knowledge proofs (e.g., verifiable shuffles) to ensure that an adversary cannot drop messages [25]. This approach reduces the number

of noise messages, but leads to significant CPU overheads due to cryptography.

Karaoke's insight is that verifiable shuffles are overkill: it is not necessary for all messages to be preserved, and it is not necessary to prove this fact to arbitrary servers. Instead, to achieve privacy, it suffices for each server to ensure that its noise is observed by all other servers. This can be done efficiently using Bloom filters, without having to reveal which messages are noise and which messages come from real users.

The contributions of this paper are as follows:

- The design of Karaoke, a metadata-private text messaging system that achieves an order of magnitude lower latency than prior work.
- Two techniques, optimistic indistinguishability and efficient noise verification, which allow Karaoke to achieve high performance.
- A privacy analysis of Karaoke's design that supports the use of these techniques.
- An experimental evaluation of a prototype of Karaoke.

One limitation of Karaoke is that it does not provide fault tolerance, since it requires all servers to be online. Handling server outages and denial-of-service attacks is an interesting direction for future work.

2 Related work

In this section, we compare Karaoke to prior work in two dimensions: privacy guarantees and the trade-off between scalability and server trust assumptions.

2.1 Privacy guarantees

Karaoke considers adversaries that control network links and some of the system's servers. This attacker model rules out systems based on Tor [7] such as Ricochet [3], due to traffic analysis attacks [5, 11, 18]. Loopix [20] is a recent system that delays messages and uses entropy [24] as a metric for reasoning about a user's anonymity set. However, Loopix does not provide any formal guarantees about privacy after users exchange multiple messages; it also requires users to trust a designated service provider [20: Table 1].

Some systems leak no information to the attacker, using techniques like DC-nets [28], Private Information Retrieval [1], or message broadcast [4]. Such systems provide the strongest form of privacy that users could hope for, but due to the quadratic overhead of these schemes in the number of users, their latency becomes high when supporting millions of users.

Karaoke achieves differential privacy for metadata-private messaging, much like Vuvuzela [26], Alpenhorn [15], and Stadium [25]. One key difference in Karaoke is that its design leaks no information about a user's traffic patterns in the common case, when there

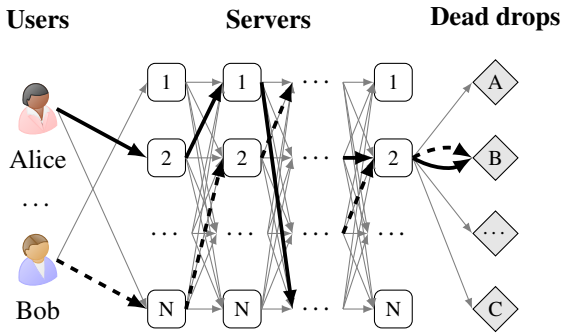


Figure 1: Overview of Karaoke's design.

are no lost messages, using the idea of optimistic indistinguishability. This allows Karaoke to add less noise for reaching the same privacy level as prior work [15, 25, 26], which improves performance.

Like Stadium, Karaoke is distributed over many machines, and must ensure that malicious servers do not compromise privacy. Stadium uses zero-knowledge proofs (e.g., verifiable shuffles) for this purpose, whereas Karaoke relies on more efficient Bloom filter checks.

2.2 Scalability vs. trust assumptions

Systems that assume the anytrust model (where all but one server may be malicious), such as Vuvuzela [26], Dissent [28], and Riposte [4], do not scale horizontally and cannot support the same magnitude of users as Karaoke.

One approach to horizontal scalability in metadata private messaging systems is to route messages through only a subset of all servers in the network, as in Loopix, Stadium, and Atom [14]. This requires trusting multiple servers to be honest, and introduces a tradeoff between the number of trusted servers (translating into the number of servers that process each message) and performance.

In Loopix every message is processed by a small number of servers (e.g., Loopix considers 3 or more servers to be a good choice [20: §4.3.1]). For privacy, Loopix requires that one of these servers is honest. However, if a significant fraction of servers are malicious, using a small number of servers means some users' messages will not be processed by any honest server. Karaoke ensures privacy with high probability by sending messages through more servers (e.g., 14 servers).

Atom [14] assumes that a fraction of the servers might be corrupt, and requires each message to be processed by many servers (hundreds). This leads to high latency, from 30 minutes to several hours. Karaoke also assumes that some fraction of servers are malicious. However it arranges its servers in a different, full-mesh topology, which allows it to achieve privacy while processing each message at fewer servers (e.g., 14 servers).

3 Overview

Figure 1 shows the main components of Karaoke. At the highest level, Karaoke consists of users, servers, and dead drops, similar to Vuvuzela and Stadium. All communication in Karaoke happens in rounds. In each round, users communicate by sending and receiving messages to and from dead drops. A dead drop is a designated location used to exchange messages. Dead drops are named by the server on which they are located, along with a pseudorandom identifier, and are not reused across rounds. When two users access the same dead drop, their messages are exchanged, and each user receives the other user's message. When two users want to communicate, they arrange to access the same dead drop (based on a shared secret). If a user is not communicating with anyone, he or she sends cover traffic to a randomly chosen dead drop.

The middle of the figure shows Karaoke's servers, labeled 1 through N, which are used to shuffle messages in order to hide information about which user is accessing which dead drop. The servers shuffle messages in *layers*, which are indicated by vertical groups in Figure 1, similar to a parallel mixnet [6, 8, 12, 21]. Each layer decrypts the messages (which are onion-encrypted) and re-orders them, so that the order of messages sent by a server does not correlate with the order in which the messages were received. Each server takes part in each layer; the figure depicts this by including each server in each layer. Between layers, servers exchange messages with one another.

The path of a message through the layers is chosen by the message sender at random. The message is onion-encrypted using the public keys of the servers on the chosen path, so that the message cannot be decrypted unless it passes through those servers. This ensures that an adversary cannot bypass the shuffling of the honest servers on the path of a message. Karaoke assumes that users know the public keys of all servers.

In Figure 1, Alice and Bob are communicating in a particular round. Their dead drop access paths are shown using bold arrows; solid for Alice and dashed for Bob. Alice and Bob send their messages to the same dead drop B on server 2. When the messages arrive at server 2, the server swaps them, and sends them back through the layers: Alice's message back to Bob along the reverse of the dashed arrows, and Bob's message back to Alice along the reverse of the solid arrows. This ensures server 2 does not know whose messages it swapped.

3.1 Goals and threat model

Karaoke's goal is to hide the communication patterns between users, so that an adversary cannot determine which users are communicating with one another. Karaoke does not hide information about which users are using Karaoke; an adversary can determine that a user is using Karaoke by observing a connection to one of Karaoke's servers. However, we hope that supporting a large number of users makes the mere act of using Karaoke less suspicious, similar to the argument by Dingledine et al. [7]. Karaoke also does not make availability guarantees; defending against DoS attacks is an interesting direction for future work.

In addition to Karaoke's privacy goals, Karaoke aims to achieve low latency for many users. This is important in order to enable broad adoption of Karaoke's design. Furthermore, Karaoke's goal is to provide horizontal scalability, so that Karaoke's operators can scale to more users over time by adding physical machines, thereby spreading the CPU and bandwidth requirements for operating Karaoke across more servers.

Karaoke assumes that an adversary has full control over the network and has compromised some number of servers and users' computers. Karaoke assumes that some fraction of servers (e.g., 80%) remains honest (not compromised), which we believe is achievable given leaked documents [19] and measurements of the Tor network [23, 27]. Karaoke hides communication patterns between users whose computers have not been compromised. If an adversary compromises a user's computer, the adversary can directly observe that user's activity, and Karaoke cannot provide any privacy guarantees. Karaoke makes standard cryptographic assumptions (the adversary cannot break cryptographic primitives), and assumes that Karaoke clients know the public keys of Karaoke servers.

We capture Karaoke's goal of hiding communication patterns using differential privacy [10], as in Vuvuzela and Stadium. Specifically, for a pair of users (call them Alice and Bob), Karaoke considers the probabilities of the observations that an adversary could make (e.g., observations of network traffic and observations from compromised servers), conditioned on Alice and Bob communicating or not communicating. Karaoke's differential privacy guarantee says that the probabilities of Alice and Bob communicating or not communicating, based on what the adversary observed, are close, and the ϵ and δ parameters control the degree of closeness (e^{ϵ} is a multiplicative factor and δ is an additive factor). The choice of the parameters is discussed in §6.1. Using differential privacy, Karaoke ensures that two users can always plausibly deny that they were communicating.

Since differential privacy is composable, a user can leverage this guarantee to reason about other plausible "cover stories." For example, if Alice was actually talking to Bob, she could instead claim she was talking to Charlie: the probability of her talking to Bob is within (ϵ, δ) of her not talking to anyone, which in turn is within (ϵ, δ) of her talking to Charlie, for a total of $(2\epsilon, 2\delta)$.

More formally, Karaoke treats the scenarios of two users communicating or not communicating with one another as "neighboring databases" in the context of dif-

ferential privacy. Since Karaoke relies on cryptography, Karaoke achieves *computational* differential privacy [17], rather than the perfect information-theoretic definition.

Karaoke's information leakage mostly comes from situations when a user's message is lost. This can occur either due to an active attack, or due to a long network outage (from which TCP cannot recover). Karaoke provides differential privacy for many rounds of message loss (hundreds, as discussed in §6.1). We expect users to avoid private conversations on highly unreliable networks; §7.6 provides some evaluation of network reliability.

Karaoke's design assumes that users can initiate conversations out-of-band. In other words, Karaoke hides metadata during a conversation. A complete messaging system would use Karaoke alongside a "dialing" protocol for one user to initiate a conversation with another user, and to establish a shared secret that is used to agree on a pseudorandom sequence of dead drops. The bootstrapping protocol would impose additional bandwidth and CPU costs for clients, but these costs are amortized over many conversation rounds. Alpenhorn [15] could serve as such a dialing protocol.

3.2 Privacy approach

Karaoke's design reveals two potential sources of information to the adversary: information about dead drop access patterns and information about how many messages were sent between servers across layers. In the rest of this section, we outline Karaoke's approach to hiding this information from the adversary.

Optimistic indistinguishability. To prevent the adversary from learning information based on dead drop access patterns, Karaoke's design strives to ensure that the dead drop access patterns look the same regardless of the communication pattern between users. Specifically, Karaoke requires that users always send *two* messages in a round. This allows a user to communicate with themselves if they are not otherwise communicating with a buddy, by arranging for their two messages to access the same dead drop. This gives the appearance of an active conversation to an adversary that is observing dead drop access patterns. If the user is communicating with a buddy, the user simply arranges for each of their messages to swap with a message from the buddy, using two different dead drops.

When the adversary is passive and there are no network outages, dead drop access patterns reveal no metadata about the communication of any pair of users. This is because, for a pair of users that might be either idle or chatting, there will be two dead drops, each of which is accessed twice. If messages are lost, an adversary may observe a dead drop with a single access, which may reveal some information. Karaoke addresses this through the use of noise messages, which we describe shortly. However, message loss is detectable in Karaoke because a user can simply look at the messages they receive back from the server to determine if any of their messages (or their buddy's messages) were lost.

Karaoke's "leakage-free" rounds allow it to improve performance by reducing noise and letting a client application decide how to handle leaky rounds. For example, the client application could choose to:

1. Alert the user, who could ignore it if their current conversation is not sensitive, or end the conversation if it is.

2. Retry the conversation after waiting (i.e., stopping the conversation but continuing to send cover traffic). This limits how quickly active attacks can learn information about the user.

3. Retry the conversation after switching to a new network (hopefully, one that is not under active attack).

These policies (or combinations of them) limit the rate at which an adversary can learn information through active attacks. This allows Karaoke to add less noise while still providing meaningful privacy guarantees.

Message swaps. A passive adversary in Karaoke can observe the number of messages sent between any two servers. To ensure that these observations do not reveal user metadata, Karaoke's topology is designed so that, for any pair of messages that traverse the same honest server in the same layer, an adversary cannot determine which path prefixes (i.e., paths leading up to this honest server) correspond to which path suffixes (i.e., paths taken by the messages after this honest server). In other words, the real scenario is indistinguishable from a scenario where the messages swap paths after the honest server.

The swapped paths correspond to the two neighboring databases. If Alice and Bob are communicating, then swapping the path suffix of one of Alice's messages with Bob's would mean that the two messages from Alice/Bob actually reach the same dead drop (so they are idle). Similarly, if Alice and Bob are idle, swapping path suffixes of two of their messages would mean that they are communicating.

This technique keeps the number of messages on each link identical regardless of whether message paths were swapped, thus preventing the adversary from learning useful information given the number of messages on every link.

Noise messages. Karaoke uses noise for two purposes: to protect dead drop access patterns in case messages are lost, and to enable message swaps. The noise takes the form of additional messages generated by the servers themselves. Each server generates messages to random dead drops, and routes those messages through random

paths in Karaoke's topology. These noise messages directly obscure the information available from the dead drop access patterns, because accesses by real users are now indistinguishable from accesses by noise messages.

Efficient noise verification. Some servers may be controlled by the adversary. It is crucial that these adversarial servers cannot subvert Karaoke's noise, either by generating insufficient noise in the first place, or by dropping noise messages as they traverse Karaoke's topology. Karaoke deals with the first problem by requiring all servers to generate enough noise to account for the possibility of malicious servers generating no noise at all.

To deal with the possibility of noise messages being dropped along the way, Karaoke uses *Bloom filters* [2] to efficiently check for the presence of noise at each layer. Each server at each layer in Karaoke's topology ensures that it has received all noise messages. It does so by computing a Bloom filter of all of the messages it has received, and sending this Bloom filter to all other servers. The other servers check whether the noise messages they generated appear in this Bloom filter. If any server indicates that their noise has been lost, the round is stopped.

Prior systems such as Stadium [25] deal with this problem by ensuring that no messages can be lost along the way. This requires expensive cryptographic techniques, such as verifiable shuffles. Karaoke's observation is that it suffices to ensure that noise messages are not lost. Using Bloom filters is a good choice because they do not require servers to reveal which messages were actually noise; the Bloom filter includes the set of all messages.

4 Design

This section describes Karaoke's design, starting with the overall structure and topology, and then describing the Karaoke client library and how Karaoke servers work.

4.1 Overall structure

Karaoke operates in rounds, which are driven by a coordinator. The coordinator is not trusted for privacy (its only job is to announce the start of a new round), but a malicious coordinator can impact the liveness of Karaoke. Round numbers must be strictly increasing, so the coordinator cannot trick clients into sending extra messages in a round, and if it announces a round multiple times, honest clients and servers will ignore it. Karaoke can distribute the user load over many coordinators (that are synchronized among themselves) since the coordinator's job is untrusted.

Karaoke's communication topology is shown in Figure 1. By using randomly chosen paths and exchanging messages at each layer, Karaoke provides a strong degree of mixing between all messages. Furthermore, Karaoke scales well with the number of servers, because each message is handled by a fixed number of servers (one per layer). As a result, adding more servers does not cause Karaoke to do more work overall.

```
def client_active(roundnum, myid, buddyid, buddysecret,
                  msg1, msg2):
  c1 = encrypt(buddysecret + "msg1" + myid, msg1)
  c2 = encrypt(buddysecret + "msg2" + myid, msg2)
  o1 = gen_onion(roundnum, myid, buddyid,
                 buddysecret + "onion1", c1)
  o2 = gen_onion(roundnum, myid, buddyid,
                 buddysecret + "onion2", c2)
  r1, r2 = karaoke_run_round(o1, o2)

  d1 = decrypt(buddysecret + "msg1" + buddyid, r1)
  d2 = decrypt(buddysecret + "msg2" + buddyid, r2)
  if d1 == None or d2 == None:
    raise("Message loss")
  return d1, d2

def client_idle(roundnum, myid):
  secret = random.secretvalue()
  c1 = random.ciphertext()
  c2 = random.ciphertext()
  o1 = gen_onion(roundnum, myid, myid + "dummy",
                 secret, c1)
  o2 = gen_onion(roundnum, myid + "dummy", myid,
                 secret, c2)
  r1, r2 = karaoke_run_round(o1, o2)
  if r1 != c2 or r2 != c1:
    raise("Message loss")

def gen_path(roundnum, rng):
  servers = get_servers_and_keys(roundnum)
  return [rng.choice(servers) for i in range(nlayers-1)]

# Choosing the last server to be one of the users' previous
# hops leads to more efficient noise generation.
def choose_last_srv(a, b):
  pair_choice = (a.id + b.id) % 2
  return sorted(a, b)[pair_choice]

def gen_onion(roundnum, myid, buddyid, secret, msg):
  mypath    = gen_path(roundnum, prng(secret + myid))
  buddypath = gen_path(roundnum, prng(secret + buddyid))
  drop_srv  = choose_last_srv(mypath[-1], buddypath[-1])
  drop_id   = prng(secret).rand128()

  onion = wrap((drop_id, msg), drop_srv)
  for srv in reversed(mypath):
    onion = wrap(onion, srv)
  return onion
```

Figure 2: Pseudocode for the Karaoke client.

4.2 Client

Figure 2 shows the pseudocode for the Karaoke client library. There are two modes of operation for the client: either the client is in an active conversation with a buddy, or the client is idle. In each round, the client must call either `client_active()` or `client_idle()`.

If the client is active, it must maintain a shared secret with the buddy, denoted `buddysecret` in the pseudocode. This secret should be established through a dialing protocol, such as Alpenhorn [15], and must evolve every round (e.g., by hashing it, or by using Alpenhorn's keywheel). Furthermore, if the client is active, it must pass two messages to `client_active()` that will be relayed to

the buddy; conversely, `client_active()` will return the buddy's two messages, if successful. Each message has a fixed size (256 bytes).

Onion generation. In each round, the client library generates two onions using `gen_onion()`. This function encapsulates a message `msg` in an onion encryption. The onion is sent towards a dead drop chosen pseudorandomly based on the shared `secret`, the ID of this user (`myid`), and the ID of the buddy (`buddyid`). For example, Figure 1's solid arrows indicate an onion sent by Alice to dead drop B on server 2. The payload, `msg`, is encrypted by the caller (specifically, by `client_active()`).

`gen_onion()` encrypts the message for each server in turn, using the public keys of the servers. The innermost encryption uses the key of the dead drop server, `drop_srv`. The other onion layers correspond to a path chosen by `gen_path()` using a pseudorandom number generator.

One subtle detail is that the dead drop server, `drop_srv`, is chosen deterministically in `gen_onion()` to be one of the servers from the two users' paths in the previous layer (either `mypath[-1]` or `buddypath[-1]`). This is an optimization that reduces the degrees of freedom in Karaoke, and thus allows Karaoke to generate noise efficiently, as we will discuss in §4.3.

The dead drop ID, `drop_id`, is chosen pseudorandomly based on the shared `secret`. This ensures that an adversary cannot learn any information by observing the accessed dead drop IDs (since the secret changes every round), yet the two users agree on the same dead drops.

Active conversation. When a client is in an active conversation, `client_active()` exchanges two messages with the user's buddy. It does so by first encrypting the two messages, `msg1` and `msg2`, to produce two ciphertexts `c1` and `c2`. The pseudocode uses + to derive subkeys from the `buddysecret` master key. `client_active()` then calls `gen_onion()` twice, with two subkeys derived from `buddysecret` (appending the strings `onion1` and `onion2` respectively). These onions are then passed to `karaoke_run_round()`, which sends the onions through Karaoke's server topology and waits for responses, if any.

Once `client_active()` receives the responses, it must verify that no message loss took place—that is, that the adversary did not block either of this user's two messages, or the buddy's two messages. `client_active()` checks for this by ensuring that it receives two ciphertexts that properly decrypt (using authenticated encryption). If an adversary dropped one of the messages from this client, `karaoke_run_round` will return `None`, causing the decryption check to fail. If an adversary dropped one of the messages from the buddy, the last server hosting the dead drop will observe just one message reaching the dead drop and echo back this client's message in response, which will similarly cause the decryption check to fail (because the message is not encrypted using the subkey generated with `buddyid`). If no message loss took place, `client_active()` returns the decrypted messages.

Sending a message back to the user in case of message loss is important since if there is a conversation between Alice and Bob, and an adversary drops Bob's message, then one naive outcome might be that now Alice receives nothing in response in that round. This would be quite unfortunate: the adversary will know Bob was talking to Alice! By echoing back the message, the last server sends at least some (fixed-size) data towards Alice, so that an adversary cannot tell that Alice was Bob's conversation partner. (To be precise, a random response would also suffice in this case.) Intermediate servers similarly enforce that every request must receive a response, in case the last server was malicious.

Idle client. When there is no active conversation, Karaoke's client library ensures that the externally observable behavior, from the adversary's perspective, remains identical. `client_idle()` does so by generating random ciphertexts, `c1` and `c2`, which should be indistinguishable from ciphertexts that would have been generated by `client_active()`. `client_idle()` chooses a random `secret`, and constructs two onions, `o1` and `o2`, simulating a conversation between users `myid` and `myid+"dummy"`.

Much like `client_active()`, `client_idle()` needs to check for message loss. It does so by ensuring that it receives `c2` and `c1` respectively in response to its onions.

Handling message loss. In Karaoke, message loss can leak information to an adversary, and thus reduce the degree of privacy that the user can expect. Karaoke detects such events, which allows the client application built on top of the library from Figure 2 to avoid excessive privacy loss. Specifically, Karaoke's client closes any active conversation after encountering message loss. This prevents an adversary from dropping a user's messages in many rounds to learn additional information. Other policies for dealing with message loss can be implemented that balance usability and privacy, as outlined in §3.

Karaoke should rarely lose messages, because IP packet loss in the network is handled by TCP (see §7.6). Thus, the primary source of false positives are long-lived network outages. We recommend that users stop sensitive conversations when their network becomes unreliable (regardless of whether it is the result of an attack).

4.3 Server

Figure 3 presents the pseudocode for Karaoke's server. The pseudocode focuses on the processing of onions from clients to the dead drops, as well as the generation and verification of noise messages. Not shown is the logic for setting up per-round public keys (signed with a long-term private key of each server), accepting inputs from users in

```
def process_layer(roundnum, layer, inputs):
  msgs = [decrypt(srvkey[roundnum], msg)
          for msg in inputs]
  msgs = dedup(msgs)
  if layer == 0:
    msgs += generate_noise(roundnum)
  else:
    bloom = bloomfilter.new(inputs)
    for srv in get_servers_and_keys(roundnum):
      if srv.rpc("check_bloom", roundnum,
                 layer, bloom) != True:
        raise("Lost noise, halting round")

  outgoing = collections.defaultdict(list)
  for m in msgs:
    outgoing[m.next_hop].append(m)

  for srv, q in outgoing:
    srv.rpc("enqueue_batch_for_process_layer",
            roundnum, layer+1, shuffle(q))

def check_bloom(roundnum, layer, bloom):
  caller = get_rpc_caller()
  for m in noise msgs routed via caller at layer:
    if m not in bloom:
      return False
  return True
```

Figure 3: Pseudocode for Karaoke's server.

the first layer, exchanging the messages that are addressed to the same dead drop in the last layer, and sending the responses back to the clients.

Layer processing. Each server uses the `process_layer()` function shown in Figure 3 to process the set of input messages at a given layer. In the first layer, the server collects input messages from clients until the round coordinator kicks off the round processing. In subsequent layers, each server waits to receive inputs from every server in the previous layer.

Layer processing starts by decrypting the inputs and de-duplicating them. It is important to remove duplicates (and to ensure the ciphertexts are not malleable), because otherwise an adversary could tag a victim's message by replicating it several times and looking for which message appears to be replicated at the end of Karaoke's topology.

Noise. The next step of layer processing involves ensuring that the necessary noise is present. In the first layer, each server generates noise; subsequent layers use Bloom filter checking to ensure that noise has not been dropped by malicious servers.

Noise generation. At the start of every round, each server generates noise. The goal of noise messages is to mask dead drop access patterns in the case of message loss, meaning that legitimate user messages did not form a pair of accesses to the same dead drop. In this case, an adversary observes some number of dead drops with two accesses, and some number with just a single access (due to a non-paired message). This translates into the two kinds of noise messages generated by Karaoke: "singles" (noise message that generates a single dead drop access), and "doubles" (a pair of noise messages that generates a double access to the same dead drop).

Karaoke's threat model assumes that some servers may be malicious, but it is not known *a priori* which servers are malicious. An adversary could use a malicious server to trace back the source of a dead drop access to the last honest server in the path. Thus, as we show in our analysis [16], it is important that all outgoing links from every server carry an adequate number of noise messages, since every link could potentially be the outgoing link from the last honest server on some message's path.

Like Stadium [25], Karaoke uses the Poisson distribution to sample noise messages. This distribution is a good fit for distributed noise generation for two reasons. First, it allows precisely sampling a non-negative integer for the number of messages, even if the distribution mean is low. Second, the sum of many small Poisson samples is also a Poisson distribution, simplifying the analysis.

Let N be the number of servers, and l be the length of Karaoke paths (`nlayers` in the pseudocode). Our topology provides N^l possible routes, which makes it computationally cumbersome to sample for every route individually, and inefficient, since there are only $(l-1)\cdot N^2$ communication links in the entire system (there are $l-1$ transitions between layers, and in each transition each server is connected to all others). We would ideally like to just sample the amount of noise on every link.

To generate the singles noise, a server begins by sampling the noise for the links to the last layer of servers (layer l), and samples how many messages go over each of the N^2 links in that phase. For each link, the server samples from the Poisson distribution, with mean λ_1. The server then sums them up to find how many of its noise messages need to leave each server in the previous layer. The server then samples again, to decide how many noise messages travel on each link to the servers in the previous layer $(l-1)$. Of course, there will likely be a mismatch; i.e., a server in layer $l-1$ has to distribute a different number of messages than it receives. In this case, the server just adds incoming or outgoing noise messages to match the other by adding extra noise messages and distributing them uniformly among all links. Karaoke continues in this fashion until it reaches the first layer. The number of these extra messages is unlikely to be large, because it is simply the difference between two samples from the same Poisson distribution. Overall, each server samples $(l-1)\cdot N^2$ times from the noise distribution to assign single-access noise.

To generate doubles noise, the server performs a similar procedure to the one described above. Notice that in the last layer we only iterate over the $N^2/2$ possible pairs of links that output messages to the same dead-drop hosting

server ($N^2/2$ is the number of possible second-to-last-hop pairs of servers, since order does not matter). This is because the dead-drop hosting server is chosen deterministically by `gen_onion()` based on `choose_last_srv()`. Similarly to the above, for each such pair, we sample noise from the Poisson distribution with mean λ_2. The result denotes the number of pairs of messages, where one message is routed on each link. In all layers before the last one, the procedure for generating double-access noise is exactly the same as the single-access noise case described above.

Preserving noise. In layers after the first one, the servers must ensure that noise messages have not been dropped by a malicious server from a previous layer. Karaoke servers do this by computing a Bloom filter [2] over all of the messages received by that server in a particular layer. Each server then sends its Bloom filter to all other servers to check whether their noise appears to be present. As long as all servers indicate that their noise is present, this server can assume that no noise messages from honest servers have been dropped, and proceed with processing the layer.

The only queries that matter are an honest relay checking with an honest noise-sender. A malicious noise-sender does not matter since it can send zero noise. A malicious relay does not matter since it can relay messages even if noise is missing. We incorporate both of these in determining how much noise is needed (generating extra noise to account for malicious servers that generate zero noise).

At each hop, one encryption layer of the message is decrypted. If an adversary does not know a server's private key, the adversary cannot predict the decryption result (it looks pseudorandom, since the onion contains another encrypted message). A malicious server that refuses to forward a message cannot guess the decrypted version of that message after the next honest hop. Thus, the adversary cannot fill in another message that will "look like" the dropped message in the Bloom filters of subsequent honest servers. Karaoke's topology and parameters ensure at least two honest servers in every path (with high probability); see analysis in §5.

To check whether noise messages are present, a server runs `check_bloom()`. This function must first determine which noise messages were routed through the calling server at a given `layer`, and second, determine the ciphertext representation of the onion that would be seen by that server at that layer. Finally, `check_bloom()` verifies that all of those ciphertexts are in the Bloom filter, without disclosing which messages are noise and which are real.

The Bloom filter has false positives, which may lead `check_bloom()` to falsely conclude that a noise message is present. In Karaoke, it is up to the server running `process_layer()` to construct the Bloom filter with adequate parameters to achieve suitably false positive rate. If the server running `process_layer()` is malicious, it can construct a Bloom filter with 100% false positive rate. However, such a malicious server could also ignore the result of `check_bloom()` altogether.

The probability of not detecting n discarded noise messages shrinks exponentially with n, since messages are independently pseudorandom (see above). This allows Karaoke to use relatively small Bloom filters (with 10% false positive rate) and yet ensure that no more than a few noise messages may be lost (for $n = 20$ the probability of missing detection is 10^{-20}). Karaoke generates a few extra noise messages to account for the possibility that several might be lost without detection (but not more).

Noise verification involves an all-to-all communication, but does not lead to quadratic bandwidth requirements as the number of servers grows. This is because increasing the number of servers would proportionally reduce the size of the Bloom filters, since the Bloom filters represent only those messages that are handled by a particular server. Other horizontally scalable systems have similar phases. For example, Stadium [25], which most closely related to Karaoke, includes an all to all distribution between "input chains" to "output chains"; in Stadium, this phase involves cryptographic computations (signature verification and NIZKs). Although in Karaoke the all-to-all communication happens at every hop, the number of hops is fixed so the overhead of Karaoke is expected to remain much smaller than Stadium even for large deployments.

5 Analysis

This sections shows that Karaoke achieves its privacy goal (§3.1), which is captured by the following theorem.

Theorem 1. Karaoke is ϵ, δ-differentially private with respect to the following neighboring databases: (1) Alice is talking with another user Bob, and (2) Alice is idle.

Proof sketch. We show Theorem 1 holds in the analysis below by the following argument. We begin by showing that Karaoke servers maintain noise messages in the system (§5.1). Next, we analyze optimistic indistinguishability, showing that in the common case Karaoke leaks no communication metadata under passive attacks (§5.2). Optimistic indistinguishability has one caveat: the attacker may launch active attacks to learn some information about the communication patterns of some users. We use differential privacy to reason about the amount of information leaked to the attacker under this scenario (§5.3).

The differential privacy parameters (ϵ and δ), the singles and doubles noise (λ_1 and λ_2), and the number of rounds k for which this theorem holds are discussed in §6.1. An extended technical report [16] provides detailed proofs.

5.1 Efficient noise verification

For Karaoke's privacy guarantees to hold, it is crucial to prevent the attacker from discarding noise messages generated by the honest servers. Karaoke identifies when noise messages are discarded using Bloom filter checks (§4.3). Bloom filters, however, allow for false positives, so a few noise messages might be dropped even if the Bloom filter check shows they are present. With a false positive rate p, the probability that k lost noise messages go undetected is p^k. Even with a relatively high $p = 10\%$, it is sufficient to increase the mean of the single- and double-access noise distributions (λ_1 and λ_2, from §4.3) by just $\frac{20}{h}$ (where h is the number of honest servers) to ensure Karaoke keeps adequate noise with probability $> 1 - 10^{-20}$.

Adjusting the Bloom filter size allows Karaoke to control the false positive rate, but the size of the Bloom filter reveals the number of messages processed by a server. This is acceptable, as the rest of Karaoke's analysis does not rely on the total number of messages being hidden.

5.2 Optimistic indistinguishability

We continue our analysis by showing that combining noise with Karaoke's routing topology prevents metadata leakage. That is, if the two messages from Alice and the two messages from Bob route through the system, then it is very likely to be completely indistinguishable whether they exchange messages with each other (active mode) or with themselves (idle mode). We begin our analysis by explaining the conditions under which optimistic indistinguishability holds, and then evaluate the probability for these conditions to hold considering a passive adversary.

5.2.1 Avoiding metadata leakage

Karaoke's optimistic indistinguishability stems from the following theorem:

Theorem 2. Assume that two messages a and b, from honest senders (users or servers), route through an honest server s^i at layer i. Denote the two message routes by $\langle s_a^1, \ldots, s^i, \ldots, s_a^l \rangle$ and $\langle s_b^1, \ldots, s^i, \ldots, s_b^l \rangle$. Then it is equally likely, given the attacker's observations of the inter-server links and malicious intermediary servers (i.e., observations on all but the last server), that a routes through $\langle s_a^{i+1} \ldots, s_a^l \rangle$ and b routes through $\langle s_b^{i+1} \ldots, s_b^l \rangle$ or vice-versa.

Proof. Since s_i is honest, its shuffle permutation is unknown to the adversary. Each message in Karaoke takes an independent route. Denote the outgoing links from server s^i that a and b take by l_1, l_2, and the attacker's observations on outgoing links from s^i by O. It holds that $\Pr[a \text{ takes } l_1 \mid O] = \Pr[b \text{ takes } l_1 \mid O]$ and that $\Pr[a \text{ takes } l_2 \mid O] = \Pr[b \text{ takes } l_2 \mid O]$. Therefore,

$$\Pr[a \text{ takes } l_1 \wedge b \text{ takes } l_2 \mid O] = \Pr[a \text{ takes } l_2 \wedge b \text{ takes } l_1 \mid O]$$

Furthermore, since messages are onion-encrypted, the bit-level representations of messages a and b forwarded by s^i are indistinguishable from random. As a result, an adversary cannot distinguish whether a travels over the link $s^i \to s_a^{i+1}$ and b over $s^i \to s_b^{i+1}$ or vice-versa.

Assume that a and b swap the suffix of their routes following layer s^i. Since the two messages swap routes, the number of messages on each following link remains the same (and the messages themselves are indistinguishable from one another because they are onion-encrypted). Therefore all of the attacker's observations on inter-server links remain the same, regardless of whether the two messages were swapped. □

Theorem 2 allows us to swap between two messages. However, it requires that the two swapped messages route through the same honest server. The next theorem, which follows from Theorem 2, extends this observation and shows that even messages with non-intersecting routes can be indistinguishably swapped, with the help of noise messages.

Theorem 3. Let a and b be two messages that route through $\langle s_a^1, \ldots, s_a^l \rangle$ and $\langle s_b^1, \ldots, s_b^l \rangle$ respectively. Let n_0 and n_1 be two other messages from honest participants that route through $\langle s_{n_0}^1, \ldots, s_{n_0}^l \rangle$ and $\langle s_{n_1}^1, \ldots, s_{n_1}^l \rangle$. Assume that there exists some i_0 and j_1 such that $s_{n_0}^{i_0} = s_a^{i_0}$ and $s_{n_0}^{j_1} = s_b^{j_1}$, where the servers $s_a^{i_0}$ and $s_b^{j_1}$ are honest and $i_0 < j_1$. This means that, for some layer i_0, n_0 and a route through the same honest server, and for some layer j_1, n_0 and b route through the same honest server. Similarly, assume there exists some i_1 and j_0 such that $s_{n_1}^{i_1} = s_b^{i_1}$ and $s_{n_1}^{j_0} = s_a^{j_0}$, where the servers $s_b^{i_1}$ and $s_a^{j_0}$ are honest, $i_1 < j_0$, $i_0 < j_0$, and $i_1 < j_1$. Under these conditions, and using observations from network links and intermediary servers, it is indistinguishable whether the messages took their actual routes or the following alternative routes:

a routes via $\langle s_a^1, \ldots, s_a^{i_0}, s_{n_0}^{i_0+1}, \ldots, s_{n_0}^{j_1-1}, s_b^{j_1}, \ldots, s_b^l \rangle$
b routes via $\langle s_b^1, \ldots, s_b^{i_1}, s_{n_1}^{i_1+1}, \ldots, s_{n_1}^{j_0-1}, s_a^{j_0}, \ldots, s_a^l \rangle$
n_0 routes via $\langle s_{n_0}^1, \ldots, s_a^{i_0}, s_a^{i_0+1}, \ldots, s_a^{j_0-1}, s_{n_1}^{j_0}, \ldots, s_{n_1}^l \rangle$
n_1 routes via $\langle s_{n_1}^1, \ldots, s_b^{i_1}, s_b^{i_1+1}, \ldots, s_b^{j_1-1}, s_{n_0}^{j_1}, \ldots, s_{n_0}^l \rangle$

Proof. Applying Theorem 2 four times on the following arguments gives the result:

1. on messages a, n_0 at honest server $s_a^{i_0}$
2. on messages b, n_1 at honest server $s_b^{i_1}$
3. on messages a, n_1 at honest server $s_a^{j_0}$
4. on messages b, n_0 at honest server $s_b^{j_1}$

Figure 4 illustrates these four swaps (where message $a = a_1$ and message $b = b_0$). □

Given four messages a, b and n_0, n_1 the attacker cannot identify, using observations on network links and malicious intermediary servers, whether the messages take one route where a, b end up on servers s^l_a, s^l_b and n_0, n_1 end up on servers $s^l_{n_0}, s^l_{n_1}$ or they take an alternative route where a, b reach s^l_b, s^l_a and n_0, n_1 reach $s^l_{n_1}, s^l_{n_0}$. However, if the last servers (s^l_*) turn out to be malicious, then the attacker might still distinguish between the two scenarios. To see why, consider the case where n_0 is a double-access noise message and its pair routes through an all-malicious route. In this case, the attacker can observe the difference between the two alternative scenarios because the last server on n_0's route would have actually received n_1 instead of n_0 and therefore would observe one less double access and two more single accesses if n_0 and n_1 were to swap (i.e., using the alternative routes in Theorem 3). The next theorem describes how messages between two honest users can be swapped without leaking information to the attacker, when n_0 and n_1 are single-access noise messages.

Theorem 4. If the premise for Theorem 3 holds for two user-messages a and b and two single-access noise messages n_0 and n_1, then it is indistinguishable whether a routes through $\langle s^1_a, \ldots, s^l_a \rangle$ and b through $\langle s^1_b, \ldots, s^l_b \rangle$, or a routes through $\langle s^1_a, \ldots, s^{i_0}_a, s^{i_0+1}_{n_0}, \ldots, s^{j_1-1}_{n_0}, s^{j_1}_b, \ldots, s^l_b \rangle$ and b routes through $\langle s^1_b, \ldots, s^{i_1}_b, s^{i_1+1}_{n_1}, \ldots, s^{j_0-1}_{n_1}, s^{j_0}_a, \ldots, s^l_a \rangle$.

Proof. Applying Theorem 3 shows that given just observations from network links and intermediary servers, an adversary cannot determine which message takes what route. We now focus on the last servers of each message route. Assume that they are all malicious and allow the attacker to observe the dead-drop access patterns. The last server on n_0's route, in the alternative routing scheme, would have received n_1 (after all four swaps); see illustration in Figure 4. Since n_1 and n_2 are two single access noise messages, generated by honest servers, the malicious last server would observe in both cases an encrypted message (that was encrypted by an honest server) reaching a dead drop by itself. Similarly this holds for the last server on n_1's route. The user messages a and b would both reach encrypted to a double-access dead drop (since the attacker is passive, the paired message reaches the dead drop too). So both cases are indistinguishable. □

We refer to two messages a and b for which there exists two single-access noise messages n_0 and n_1 that satisfy the premise of Theorem 4 as *indistinguishably swappable*. We next use Theorem 4 to analyze Karaoke's privacy guarantees.

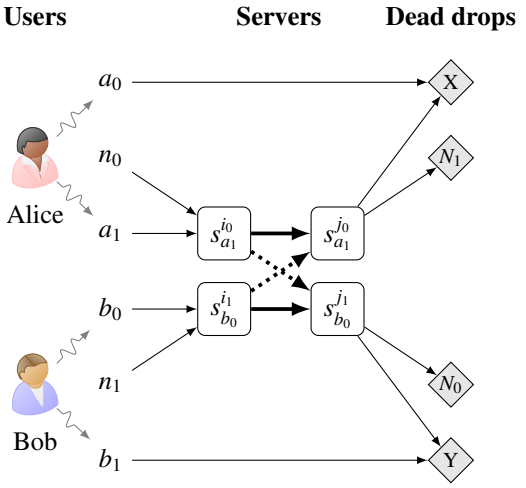


Figure 4: An illustration of Karaoke's optimistic indistinguishability: an adversary cannot determine whether Alice and Bob are communicating via dead drops X and Y. Straight lines represent links (potentially across multiple intermediate servers) that an adversary can track. Servers $s^{i_0}_{a_1}$, $s^{i_1}_{b_0}$, $s^{j_0}_{a_1}$, and $s^{j_1}_{b_0}$ are honest. Solid bold lines indicate the actual path taken by messages a_1 and b_0. Dotted bold lines indicate the actual path taken by messages n_0 and n_1. An adversary cannot distinguish whether a_1 and b_0 took the solid or dotted bold lines. Squiggly lines indicate users generating two messages in a round.

5.2.2 Alice talking with Bob, and claims "idle"

Consider two users, Alice and Bob, who may be talking with each other or idle. Alice sends two messages a_0, a_1 and Bob sends b_0, b_1. If Alice and Bob communicate, then Alice's a_0 meets Bob's b_0 at the dead drop, and a_1 meets b_1 at a different (and independently chosen) dead drop. If they do not communicate, then a_0 meets a_1 at a dead drop and so do b_0 and b_1.

Theorem 5. If one of the pairs of messages $\langle a_0, b_1 \rangle$ or $\langle a_1, b_0 \rangle$ is indistinguishably swappable, then it is indistinguishable whether Alice is talking to Bob or they are both idle.

Proof. To understand why this theorem holds, consider Figure 4. Assume without loss of generality that the premise holds for the pair of messages $\langle a_1, b_0 \rangle$. Applying Theorem 4 on $\langle a_1, b_0 \rangle$, it is therefore indistinguishable whether a_1 routes to dead drop X and b_0 routes to dead drop Y or vise versa. In the first scenario a_0 meets b_0 at dead drop Y and a_1 meets b_1 at dead drop X, so Alice and Bob are talking. In the second (indistinguishable) scenario it is actually a_0 that meets a_1 at dead drop X and b_0 that meets b_1 at dead drop Y so Alice and Bob are idle. Importantly, it does not matter what route Alice and Bob's other messages, a_0 and b_1, take; the servers handling these messages may all be malicious. □

Our technical report [16] analyzes the probability with which optimistic indistinguishability holds. For example, with $N = 100$ servers, out of which $h = 80$ are assumed

honest, a chain length of $l = 14$, and where each honest server generates single-access noise with mean $\lambda_1 \geq 0.5$ (so the mean of single-access noise on each link is $h\lambda_1 = 40$), the probability that optimistic indistinguishability holds is at least $1 - 5 \cdot 10^{-14}$.

5.2.3 Alice idle, and claims "talking with Bob"

Theorem 6. If the premise for Theorem 4 holds for at least one of the message pairs $\langle a_0, b_0 \rangle$, $\langle a_0, b_1 \rangle$, $\langle a_1, b_0 \rangle$, $\langle a_1, b_1 \rangle$, then it is indistinguishable whether Alice is talking to Bob or they are both idle.

When Alice and Bob are idle, a_0, a_1 and b_0, b_1 travel to the same dead drop. It is therefore sufficient to indistinguishably swap one of four options: a_0 with b_0, or a_0 with b_1, or a_1 with b_0, or a_1 with b_1 (rather than two options as in §5.2.2: a_0 with b_1, or a_1 with b_0). This gives an even higher probability of achieving indistinguishability.

5.3 Message loss and differential privacy

An active attacker can discard user messages before Karaoke unlinks them from their senders (e.g., before the first layer, as users submit messages to Karaoke). This might prevent Karaoke from "indistinguishably swapping" messages as required for our analysis in the passive case (§5.2). We now analyze this scenario. The technical report [16] includes the proofs for the theorems below.

Consider a user Alice and an active attacker who tries to learn whether she is talking with Bob.

Theorem 7. The active attacker's best strategy (leaking the most information) is to either discard both messages from Alice, or both messages from Bob.

Intuitively, the theorem holds since if the attacker discards both messages from Alice or both messages from Bob, there are no messages to swap with so optimistic indistinguishability never holds. The following theorem holds when the attacker is active:

Theorem 8. Karaoke is ϵ, δ-differentially private in the face of message loss (e.g., due to active attackers), if both user messages route through at least two honest servers.

The conditional in Theorem 8 holds with overwhelming probability in the route length parameter l. For example, with a route length $l = 14$, assuming 80% of the servers are honest, this conditional holds with probability $1 - 2 \cdot 10^{-8}$ (which is folded into the differential privacy δ parameter of Karaoke).

6 Implementation

Karaoke is implemented in 4000 lines of Go code, compiled with Go 1.11. Onion decryption dominates the CPU costs of our prototype and is implemented in native `amd64` assembly, provided by Go's NaCl library. The servers use the gRPC library over TLS for communication. We use streaming RPCs and batching RPCs together to reduce latency. Karaoke issues RPCs over multiple TCP connections to improve throughput.

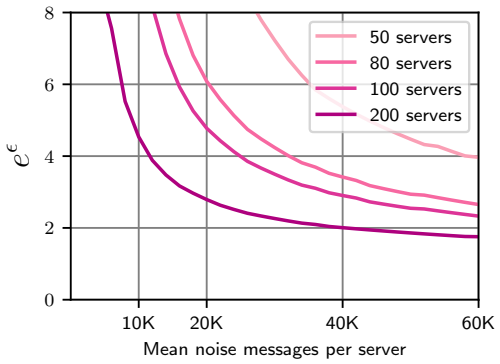


Figure 5: e^ϵ as a function of the number of noise messages per server per round, for $\delta = 10^{-4}$, $h = \lfloor 0.8N \rfloor$, and $l = 14$.

6.1 Parameter selection

We would like Karaoke to provide good privacy guarantees even after users communicate via Karaoke for a long time. We target $\epsilon = \ln 4$ and $\delta = 10^{-4}$ after 10^8 rounds of communication, of which 245 rounds encounter message loss during a sensitive conversation.

Figure 5 plots the expected number of noise messages that an honest server generates in a round, and the resulting e^ϵ privacy guarantee (with a fixed $\delta = 10^{-4}$ after 10^8 communication rounds with 245 rounds of message loss), for deployments of $N = 50, \ldots, 200$ servers where we assume $h = \lfloor 0.8N \rfloor$ servers are honest, and route length $l = 14$. For example, in our configuration using 100 servers, each server generates an average of $N^2\lambda_1 + N^2\lambda_2 = 25$K noise messages per round. Computing the data in Figure 5 required the use of composition over multiple rounds [10, 13].

As we evaluate in §7.6, 245 rounds of message loss is about an order of magnitude higher than the number of expected losses due to network outages in a year. Karaoke could achieve the same privacy guarantee under more active attacks by adding more noise.

7 Evaluation

We quantitatively answer the following questions:

- Can Karaoke achieve low latency for many users?
- Can Karaoke scale to more users by adding servers while maintaining the same low latency?
- How is Karaoke's performance affected by the fraction of honest servers?
- How important are Karaoke's techniques for achieving low latency?

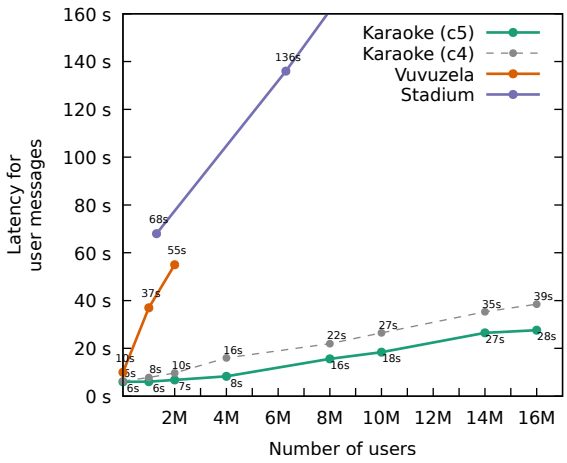


Figure 6: End-to-end latency of user messages with a varying number of users. Vuvuzela is running with 3 servers; Karaoke and Stadium are both running with 100 servers.

- How often would network problems cause Karaoke users to observe message loss?

7.1 Experimental setup

To answer the above questions we ran our prototype on Amazon EC2 using `c5.9xlarge` instances (36× Intel Xeon 3.0 GHz cores with 72 GB of memory and 10 Gbps links). We ran experiments using VMs in the same data center to save on AWS bandwidth costs. Realistically, Karaoke would be deployed on servers in different countries (or trust zones). For example, we envision some fraction of the servers running in the US and the rest running in different countries in Europe. We simulate this topology by adding 100ms of round-trip network latency (the round-trip time from the east coast of the US to Europe) to each VM using the `tc qdisc` command.

We simulate millions of users by having servers generate extra messages in the first layer (to avoid the cost of launching many more client VMs). The extra messages are pre-generated (before the round starts) so that server CPU costs are not muddled by what would normally be client CPU costs.

An additional VM is used to run a coordinator server. This server has two jobs: it starts rounds across all Karaoke servers and injects probe messages into each round to measure the end-to-end latency of the round.

Unless specified otherwise, our experiments assume that 80% of the servers are honest, which translates into a topology with 14 layers. Karaoke's Bloom filters are tuned for a 10% false positive rate, as discussed in §5.1.

7.2 Karaoke achieves low latency

To evaluate Karaoke's end-to-end latency we ran an experiment using 100 Karaoke servers. Figure 6 shows the results. For comparison, we also include the latency of Vuvuzela and Stadium as reported in their papers which provide privacy comparable to Karaoke. The Vuvuzela and Stadium results used `c4.8xlarge` VMs, so we also measured Karaoke's performance on this less powerful instance type. Stadium's performance was achieved using 100 servers with a chain length of 9. Vuvuzela used only 3 servers because its performance does not increase with the number of servers.

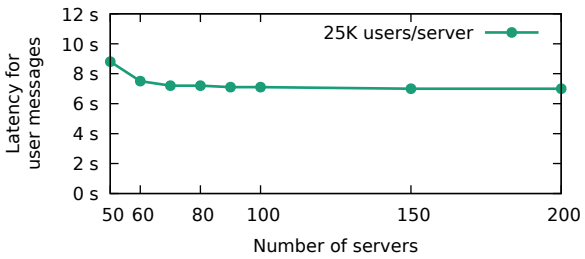


Figure 7: End-to-end latency of user messages with 25K users per server, with a varying number of servers.

The results show that with 2M users Karaoke achieves 5× lower latency than Vuvuzela, and 8× lower latency than Stadium (using the weaker `c4` instances). Furthermore, the slope of the Karaoke line in Figure 6 shows that Karaoke scales better with more users than either Vuvuzela or Stadium. Karaoke's scaling is better than Vuvuzela because only a fraction of Karaoke servers are involved in handling the messages from every additional user, whereas every Vuvuzela server must handle every additional user's messages. Karaoke's scaling is better than Stadium because Stadium must perform expensive zero-knowledge proofs for every additional user message, whereas Karaoke's marginal cost are just in onion decryption and network bandwidth. For instance, Karaoke achieves 10× lower latency than Stadium with 16M users.

7.3 Scaling by adding servers

The previous subsection shows that Karaoke's latency increases as more users join the system. This is unavoidable if the number of servers is fixed. Ideally, Karaoke would be able to support additional users without increasing latency by adding a proportional number of servers. To evaluate if this is the case, we measured the end-to-end latency of Karaoke with a varying number of servers and a proportional number of users (25K users per server).

Figure 7 shows the results, which indicate that Karaoke can maintain low latency for an increasing number of users by adding more servers to the system. Karaoke's latency goes down slightly as the number of servers grows because it requires less noise, as shown in Figure 5.

7.4 Fraction of honest servers

Figure 8 shows the number of layers required to achieve Karaoke's privacy guarantees with a varying fraction of honest servers, and the impact that increasing the number of layers has on end-to-end latency. The results show Karaoke's tradeoff between lower latency and fewer trusted servers. When fewer servers are assumed honest,

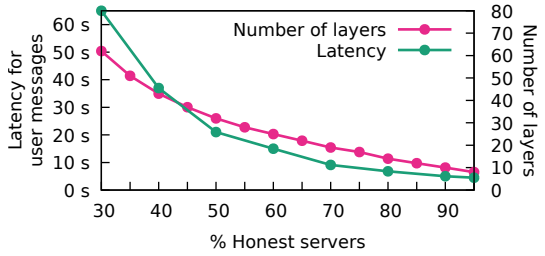


Figure 8: End-to-end latency for 2M user messages and 100 servers with a varying fraction of honest servers. The right y-axis shows the required number of layers to achieve privacy for a given fraction of honest servers.

each honest server has to create more noise to compensate for the possibility of malicious servers not sending any noise. Karaoke achieves acceptable latency for text messaging even if only 60% of the servers are honest. On the other hand, Karaoke would not be a good fit if only 30% of the server were honest.

7.5 Importance of techniques

To demonstrate the importance of Karaoke's key techniques (optimistic indistinguishability and using Bloom filters for efficient noise verification), we consider the performance of Karaoke without these techniques. In the absence of optimistic indistinguishability, Karaoke would need to add ~320K noise messages per server per round to achieve the same level of privacy. This translates into an increase in latency from 6.8s to 31s for 2 million users.

In the absence of Bloom filters, Karaoke could use verifiable shuffles similar to Stadium. For 6 million users and 100 servers, each Stadium server spends 6s generating verifiable shuffles and another 2s verifying shuffles at each hop in the network. Karaoke, on the other hand, spends 250ms generating and checking Bloom filters at each hop. Using verifiable shuffles in Karaoke would increase Karaoke's overall latency by about 2 minutes (8 seconds for each of Karaoke's 14 hops). This shows that both techniques are crucial for Karaoke's performance.

7.6 Leakage due to network issues

Karaoke's design avoids leaking information when the network is well-behaved, by arranging for all dead drop access to occur in pairs. However, network issues could result in some information being leaked if some dead drop accesses are no longer paired. Karaoke runs over TCP so momentary packet loss will not prevent message delivery. On the other hand, if clients can not communicate with the Karaoke servers for an extended period of time, they will be unable to submit their message into a round.

To estimate how often this might happen, we performed an experiment by probing a Karaoke server every 2 minutes for a day from 100 machines using RIPE ATLAS [22], which provided machines distributed across the globe that communicate with our server. Each probe consisted of 3 ping packets, spaced 1 second apart. The experiment generated 71,194 probe results, of which 70,106 received responses to all 3 pings, 991 received 2 responses, 60 received 1 response, and 37 received no responses (indicating a complete loss of network connectivity). The complete losses of network connectivity occurred in "bursts," where a machine experienced complete loss of connectivity for several adjacent two-minute intervals. The complete losses were encountered by 8 machines (7 of them observing one "burst" and one observing two "bursts").

These results suggest that a Karaoke client could encounter approximately 9 message loss events over 100 days, or about 33 such events per year. (Since Karaoke clients switch to idle mode after detecting message loss, only the first loss in a burst matters for this analysis.) This compares favorably with the message loss that Karaoke's parameters can handle (245, as discussed in §6.1).

8 Conclusion

Karaoke improves the latency of metadata-private text messaging by almost an order of magnitude compared to prior work. Karaoke also scales well with the number of users and the number of servers, maintaining its low latency. To achieve its performance, Karaoke introduces a new design, exchanging messages between each server in multiple layers, as well as two key techniques. *Optimistic indistinguishability* allows Karaoke to achieve perfect privacy with high probability in case no messages from the user (and their peer) are lost, and allows clients to detect message loss. *Efficient noise verification* allows Karaoke to generate noise messages across many servers, and to use efficient Bloom filter checks to prevent adversaries from discarding the noise. We hope that Karaoke's low latency will bring metadata-private messaging closer to widespread adoption.

Acknowledgments

Thanks to Derek Leung, Georgios Vlachos, Adam Suhl, and the PDOS group for their helpful comments and suggestions. Thanks also to the anonymous reviewers and our shepherd, Ranjita Bhagwan. This work was supported by NSF awards CNS-1413920 and CNS-1414119, and by Google.

References

[1] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–569, Savannah, GA, Nov. 2016.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] J. Brooks et al. Ricochet: Anonymous instant messaging for real privacy, 2016. https://ricochet.im.

[4] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 321–338, San Jose, CA, May 2015.

[5] G. Danezis. Statistical disclosure attacks: Traffic confirmation in open environments. In *Proceedings of the 18th International Conference on Information Security*, pages 421–426, Athens, Greece, May 2003.

[6] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the 24th IEEE Symposium on Security and Privacy*, pages 2–15, Oakland, CA, May 2003.

[7] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[8] R. Dingledine, V. Shmatikov, and P. F. Syverson. Synchronous batching: From cascades to free routes. In *Proceedings of the Workshop on Privacy Enhancing Technologies*, pages 186–206, Toronto, Canada, May 2004.

[9] Z. Dorfman. Botched CIA communications system helped blow cover of Chinese agents. *Foreign Policy*, Aug. 2018. https://foreignpolicy.com/2018/08/15/botched-cia-communications-system-helped-blow-cover-chinese-agents-intelligence/.

[10] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[11] Y. Gilad and A. Herzberg. Spying in the dark: TCP and Tor traffic analysis. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium*, pages 100–119, Vigo, Spain, July 2012.

[12] P. Golle and A. Juels. Parallel mixing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 220–226, Washington, DC, Oct. 2004.

[13] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *Proceedings of the 32nd International Conference on Machine Learning*, Lille, France, 2015.

[14] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 406–422, Shanghai, China, Oct. 2017.

[15] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 571–586, Savannah, GA, Nov. 2016.

[16] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis (extended technical report). Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Oct. 2018. Also available at https://vuvuzela.io/karaoke-extended.pdf.

[17] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO)*, pages 126–142, Santa Barbara, CA, Aug. 2009.

[18] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, pages 183–195, Oakland, CA, May 2005.

[19] National Security Agency. Tor stinks. The Guardian, Oct. 2013. https://www.theguardian.com/world/interactive/2013/oct/04/tor-stinks-nsa-presentation-document.

[20] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The Loopix anonymity system. In *Proceedings of the 26th USENIX Security Symposium*, pages 1199–1216, Vancouver, Canada, Aug. 2017.

[21] C. Rackoff and D. R. Simon. Cryptographic defense against traffic analysis. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 672–681, 1993.

[22] RIPE Network Coordination Centre. RIPE Atlas, May 2018. https://atlas.ripe.net/.

[23] A. Sanatinia and G. Noubir. Honey onions: A framework for characterizing and identifying misbehaving Tor HSDirs. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*, pages 127–135, Oct. 2016.

[24] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *Proceedings of the Workshop on Privacy Enhancing Technologies*, pages 41–53, San Francisco, CA, Apr. 2002.

[25] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–440, Shanghai, China, Oct. 2017.

[26] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, Monterey, CA, Oct. 2015.

[27] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious Tor exit relays. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium*, pages 304–331, Amsterdam, Netherlands, July 2014.

[28] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

Obladi: Oblivious Serializable Transactions in the Cloud

Natacha Crooks[⋆†] *Matthew Burke*[†] *Ethan Cecchetti*[†]
Sitar Harel[†] *Rachit Agarwal*[†] *Lorenzo Alvisi*[†]

[⋆]*University of Texas at Austin* [†]*Cornell University*

Abstract

This paper presents the design and implementation of Obladi, the first system to provide ACID transactions while also hiding access patterns. Obladi uses as its building block oblivious RAM, but turns the demands of supporting transactions into a performance opportunity. By executing transactions within epochs and delaying commit decisions until an epoch ends, Obladi reduces the amortized bandwidth costs of oblivious storage and increases overall system throughput. These performance gains, combined with new oblivious mechanisms for concurrency control and recovery, allow Obladi to execute OLTP workloads with reasonable throughput: it comes within 5× to 12× of a non-oblivious baseline on the TPC-C, SmallBank, and FreeHealth applications. Latency overheads, however, are higher (70× on TPC-C).

1 Introduction

This paper presents Obladi, the first cloud-based key value store that supports transactions while hiding access patterns from cloud providers. Obladi aims to mitigate the fundamental tension between the convenience of offloading data to the cloud, and the significant privacy concerns that doing so creates. On the one hand, cloud services [3, 4, 48, 49, 62] offer clients scalable, reliable IT solutions and present application developers with feature-rich environments (transactional support, stronger consistency [23, 52], etc.). Medical practices, for instance, increasingly prefer to use cloud-based software to manage electronic health records (EHR) [17, 39]. On the other hand, many applications that could benefit from cloud services store personal data that can reveal sensitive information even when encrypted or anonymized [53, 54, 74, 83]. For example, charts accessed by oncologists can reveal not only whether a patient has cancer, but also, depending on the frequency of accesses (e.g., the frequency of chemotherapy appointments), indicate the cancer's type and severity. Similarly, travel websites have been suspected of increasing the price of frequently searched flights [83]. Hiding *access patterns*—that is, hiding not only the content of an object, but also when and how frequently it is accessed, is thus often desirable.

Responding to this challenge, the systems community has taken a fresh look at private data access. Recent solutions, whether based on private information retrieval [2, 31], Oblivious RAM [15, 44, 70], function sharing [83], or trusted hardware [5, 7, 25, 44, 81], show that it is possible to support complex SQL queries without revealing access patterns.

Obladi addresses a complementary issue: supporting ACID transactions while guaranteeing data access privacy. This combination raises unique challenges [5], as concurrency control mechanisms used to enforce isolation, and techniques used to enforce atomicity and durability, all make hiding access patterns more problematic (§3).

Obladi takes as its starting point Oblivious RAM, which provably hides all access patterns. Existing ORAM implementations, however, cannot support transactions. First, they are not fault-tolerant. For security and performance, they often store data in a client-side *stash*; durability requires the stash content to be recoverable after a failure, and preserving privacy demands hiding the stash's size and contents, even during failure recovery. Second, ORAM provides limited or no support for concurrency [12, 70, 75, 86], while transactional systems are expected to sustain highly concurrent loads.

Obladi demonstrates that the demands of supporting transactions can not only be met, but also turned into a performance opportunity. Its key insight is that transactions actually afford more flexibility than the single-value operations supported by previous ORAMs. For example, serializability [61] requires that the effects of transactions be reflected consistently in the state of the database *only after they commit*. Obladi leverages this flexibility to delay committing transactions until the end of fixed-size epochs, buffering their execution at a trusted proxy and enforcing consistency and durability only at epoch boundaries. This delay improves ORAM throughput without weakening privacy.

The ethos of *delayed visibility* is the core that drives Obladi's design. First, it allows Obladi to implement a multiversioned database atop a single-versioned ORAM, so that read operations proceed without blocking, as with other multiversioned

databases [10], and intermediate writes are buffered locally: only the *last* value of any key modified during an epoch is written back to the ORAM. Delaying writes reduces the number of ORAM operations needed to commit a transaction, lowering amortized CPU and bandwidth costs without increasing contention: Obladi's concurrency control ensures that delaying commits does not affect the set of values that transactions executing within the same epoch can observe.

Second, it allows Obladi to securely parallelize Ring ORAM [69], the ORAM construction on which it builds. Obladi pipelines conflicting ORAM operations rather than processing them sequentially, as existing ORAM implementations do. This parallelization, however, is only secure if the write-back phase of the ORAM algorithm is delayed until pre-determined times, namely, epoch boundaries.

Finally, delaying visibility gives Obladi the ability to abort entire epochs in case of failure. Obladi leverages this flexibility, along with the near-deterministic write-back algorithm used by Ring ORAM, to drastically reduce the information that must be logged to guarantee durability and privacy-preserving crash recovery.

The results of a prototype implementation of Obladi are promising. On three applications (TPC-C [80], SmallBank [22], and FreeHealth [42], a real medical application) Obladi is within 5×-12× of the throughput of non-private baselines. Latency is higher (70×), but remains reasonable (in the hundreds of milliseconds).

To summarize, this paper makes three contributions:

1. It presents the design, implementation, and evaluation of the first ACID transactional system that also hides access patterns.
2. It introduces an epoch-based design that leverages the flexibility of transactional workloads to increase overall system throughput and efficiently recover from failures.
3. It provides the first formal security definition of a transactional, crash-prone, and private database. Obladi uses the UC-security framework [14], ensuring that security guarantees hold under concurrency and composition.

Obladi also has several limitations. First, like most ORAMs that regulate the interactions of multiple clients, it relies on a local centralized proxy, which introduces issues of fault-tolerance and scalability. Second, Obladi does not currently support range or complex SQL queries. Addressing the consistency challenge of maintaining oblivious indices [5, 25, 89] in the presence of transactions is a promising avenue for future work.

2 Threat and Failure Model

Obladi's threat and failure assumptions aim to model deployments similar to those of medical practices, where doctors and nurses access medical records through an on-site server, but choose to outsource the integrity and availability of those records to a cloud storage service [17, 39].

Threat Model. Obladi adopts a *trusted proxy* threat model [70, 75, 86]: it assumes multiple mutually-trusting client applications interacting with a single trusted proxy in a single shared administrative domain. The applications issue transactions and the proxy manages their execution, sending read and write requests on their behalf over an asynchronous and unreliable network to an *untrusted storage server*. This server is controlled by an honest-but-curious adversary that can observe and control the timing of communication to and from the proxy, but not the on-site communication between application clients and the proxy. We extend our threat model to a fully malicious adversary in our technical report [20]. We consider attacks that leak information by exploiting timing channel vulnerabilities in modern processors [13, 36, 43] to be out of scope. Obladi guarantees that the adversary will learn no information about: (*i*) the decision (commit/abort) of any ongoing transaction; (*ii*) the number of operations in an ongoing transaction; (*iii*) the type of requests issued to the server; and (*iv*) the actual data they access. Obladi does not seek to hide the type of application that is currently executing (ex: OLTP vs OLAP).

Failure Model. Obladi assumes cloud storage is reliable, but, unlike previous ORAMs, explicitly considers that both application clients and the proxy may fail. These failures should invalidate neither Obladi's privacy guarantees nor the Durability and Atomicity of transactions.

3 Towards Private Transactions

Many distributed, disk-based commercial database systems [8, 19, 58] separate concurrency control logic from storage management: SQL queries and transactional requests are regulated in a concurrency control unit and are subsequently converted to simple read-write accesses to key-value/file system storage. As ORAMs expose a read-write address space to users, a logical first attempt at implementing oblivious transactions would simply replace the database storage with an arbitrary ORAM. This black-box approach, however, raises both security concerns (§3.1) and performance/functionality issues (§3.2)

Security guarantees can be compromised by simply enforcing the ACID properties. Ensuring Atomicity, Isolation, and Durability imposes additional structure on the order of individual reads and writes, introducing sources of information leakage [5, 72] that do not exist in non-transactional ORAMs (§3.1). Performance and functionality, on the other hand, are hampered by the inability of current ORAMs to efficiently support highly concurrent loads and guarantee Durability.

3.1 Security for Isolation and Durability

The mechanisms used to guarantee Isolation, Atomicity, and Durability introduce timing correlations that directly leak information about the data accessed by ongoing transactions.

Concurrency Control. Pessimistic concurrency controls like two-phase locking [26] delay operations that would violate serializability: a write operation from transaction T_1

cannot execute concurrently with any operation to the same object from transaction T_2. Such blocking can potentially reveal sensitive information about the data, even when executing on top of a construction that hides access patterns: a sudden drop in throughput could reveal the presence of a deadlock, of a write-heavy transaction blocking the progress of read transactions, or of highly contended items accessed by many concurrent transactions. More aggressive concurrency control schemes like timestamp ordering or multiversioned concurrency control [1, 10, 34, 41, 66, 67, 87] allow transactions to observe the result of the writes of other ongoing transactions. These schemes improve performance in contended workloads, but introduce the potential for *cascading aborts*: if a transaction aborts, all transactions that observed its write must also abort. If a write-heavy transaction T_{heavy} aborts, it may cause a large number of transactions to rollback, again revealing information about T_{heavy} and, perhaps more problematically, about the set of objects that T_{heavy} accessed.

Failure Recovery. When recovering from failure, Durability requires preserving the effects of committed transactions, while Atomicity demands removing any changes caused by partially-executed transactions. Most commercial systems [50, 58, 59] preserve these properties through variants of *undo* and *redo* logging. To guarantee Durability, write and commit operations are written to a redo log that is replayed after a failure. To guarantee Atomicity, writes performed by partially-executed transactions are *undone* via an *undo log*, restoring objects to their last committed state. Unfortunately, this undo process can leak information: the number of undo operations reveals the existence of ongoing transactions, their length, and the number of operations that they performed.

3.2 Performance/functionality limitations

Current ORAMs align poorly with the need of modern OLTP workloads, which must support large numbers of concurrent requests; in contrast, most ORAMs admit little to no concurrency [12, 70, 75, 86] (we benchmark the performance of sequential Ring ORAM in Figure 10a).

More problematically, ORAMs provide no support for fault-tolerance. Adding support for Durability presents two main challenges. First, most ORAMs require the use of a *stash* that temporarily buffers objects at the client and requires that these objects be written out to server storage in very specific ways (as we describe further in §4). This process aligns poorly with guaranteeing Durability for transactions. Consider for example a transaction T_1 that reads the version of object x written by T_2 and then writes object y. To recover the database to a consistent state, the update to x should be flushed to cloud storage before the update to y. It may however not be possible to *securely* flush x from the stash before y. Second, ORAMs store metadata at the client to ensure that cloud storage observes a request pattern that is independent of past and currently executing operations. As we show in §8, recovering this metadata after a failure can lead to duplicate accesses that leak information.

3.3 Introducing Obladi

These challenges motivate the need to co-design the transactional and recovery logic with the underlying ORAM data structure. The design should satisfy three goals: (*i*) security—the system should not leak access patterns; (*ii*) correctness—Obladi should guarantee that transactions are serializable; and (*iii*) performance—Obladi should scale with the number of clients. The principle of *workload independence* underpins Obladi's security: the sequence of requests sent to cloud storage shoud remain independent of the type, number, and access set of the transactions being executed. Intuitively, we want Obladi's sequence of accesses to cloud storage to be statistically indistinguishable from a sequence that can be generated by an Obladi *simulator* with no knowledge of the actual transactions being run by Obladi. If this condition holds, then observing Obladi's accesses cannot reveal to the adversary any information about Obladi's workload. We formalize this intuition in our security definition in §9.

Much of Obladi's novelty lies not in developing new concurrency control or recovery mechanisms, but in identifying what standard database techniques can be leveraged to lower the costs of ORAM while retaining security, and what techniques instead subtly break obliviousness.

To preserve workload independence while guaranteeing good performance in the presence of concurrent requests, Obladi centers its design around the notion of *delayed visibility*. Delayed visibility leverages the observation that, on the one hand, ACID consistency and Durability apply only when transactions commit, and, on the other, commit operations can be delayed. Obladi leverages this flexibility to delay commit operations until the end of *fixed-size epochs*. This approach allows Obladi to (*i*) amortize the cost of accessing an ORAM over many concurrently executing requests; (*ii*) recover efficiently from failures; and (*iii*) preserve workload independence: the epochs' deterministic structure allows Obladi to decouple its externally observable behavior from the specifics of the transactions being executed.

4 Background

Oblivious **R**emote **A**ccess **M**emory is a cryptographic protocol that allows clients to access data outsourced to an untrusted server without revealing what is being accessed [29]; it generates a sequence of accesses to the server that is completely *independent* of the operations issued by the client. We focus specifically on *tree-based* ORAMs, whose constructions are more efficiently implementable in real systems: to date, they have been implemented in hardware [27, 46] and as the basis for blockchain ledgers [15] with reasonable overheads. Most tree-based ORAMs follow a similar structure: objects (usually key-value pairs) are mapped to a random leaf (or *path*) in a binary tree and physically reside (encrypted) in

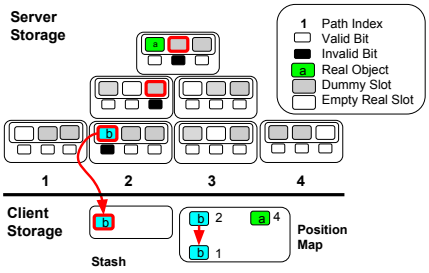


Figure 1: Ring ORAM - Read (Z=1, S=2)

some tree node (or *bucket*) along that path. Objects are logically removed from the tree and remapped to a new random path when accessed. These objects are eventually flushed back to storage (according to their new path) as part of an *eviction* phase. Through careful scheduling, this write-back phase does not reveal the new location of the objects; objects that cannot be flushed are kept in a small client-side *stash*.

Ring ORAM. Obladi builds upon Ring ORAM [69], a tree-based ORAM with two appealing properties: a constant stash size and a fully deterministic eviction phase. Obladi leverages these features for efficient failure recovery.

As shown in Figure 1, server storage in Ring ORAM consists of a binary tree of *buckets*, each with a fixed number $Z+S$ of *slots*. Of these, Z are reserved for storing actual encrypted data (*real objects*); the remaining S exclusively store *dummy objects*. Dummy objects are blocks of encrypted but meaningless data that appear indistinguishable from real objects; their presence in each bucket prevent the server from learning how many real objects the bucket contains and which slots contains them. A random permutation (stored at the client) determines the location of dummy slots. In Figure 1, the root bucket contains a real slot followed by two dummy slots; the real slot contains the data object *a*; its left child bucket instead contains dummy slots in positions one and three, and an empty real slot in second position.

Client storage, on the other hand, is limited to (*i*) a constant sized *stash*, which temporarily buffers objects that have yet to be replaced into the tree and, unlike a simple cache, is essential to Ring ORAM's security guarantees; (*ii*) the set of current *permutations*, which identify the role of each slot in each bucket and record which slot have already been accessed (and marked *invalid*); and (*iii*) a *position map*, which records the random leaf (or *path*) associated with every data object. In Ring ORAM, objects are mapped to individual leaves of the tree but can be placed in any one of the buckets along the path from the root to that leaf. For instance, object *a* in Figure 1 is mapped to *path 4* but stored in the root bucket, while object *b* is mapped to *path 2* and stored in the leaf bucket of this path.

Ring ORAM maintains two core invariants. First, each data object is mapped to a new leaf chosen uniformly at random after every access, and is stored either in the stash, or in a bucket on the path from the tree's root to that leaf (**path invariant**). Second, the physical positions of the $Z+S$ dummy and real objects in each bucket are randomly permuted with respect to all past and future writes to that bucket (i.e., no slot can be accessed more than once between permutations) (**bucket invariant**). The server never learns whether the client accesses a real or a dummy object in the bucket, so the exact position of the object along that path is never revealed.

Intuitively, the path invariant removes any correlation between two accesses to the same object (each access will access independent random paths), while the bucket invariant prevents the server from learning when an object was last accessed (the server cannot distinguish an access to a real slot from a dummy slot). Together, these invariants ensure that, regardless of the data or type of operation, all access patterns will look indistinguishable from a random set of leaves and slots in buckets.

Access Phase. The procedures for read and write requests is identical. To access an object *o*, the client first looks up *o*'s path in the position map, and then reads one object from each bucket along that path. It reads *o* from the bucket in which it resides and a valid dummy object from each other bucket, identified using its local permutation map. Finally, *o* is remapped to a new path, updated to a new value (if the request was a write), and added to the stash; importantly, *o* is not immediately written back out to cloud storage.

Figure 1 illustrates the steps involved in reading an object *b*, initially mapped to path 2. The client reads a dummy object from the first two buckets in the path (at slots two and three respectively), and reads *b* from the first slot of the bottom bucket. The three slots accessed by the client are then marked as invalid in their respective buckets, and *b* is remapped to path 1. To write a new object *c*, the client would have to read three valid dummy objects from a random path, place *c* in the stash, and remap it to a new path.

Access Security. Remapping objects to independent random paths prevents the server from detecting repeated accesses to data, while placing objects in the stash prevents the server from learning the new path. Marking read slots as invalid forces every bucket access to read from a distinct slot (each selected according to the random permutation). The server consequently observes uniformly distributed accesses (without repetition) independently of the contents of the bucket. This lack of correlation, combined with the inability to distinguish real slots from dummy slots, ensures that the server does not learn if or when a real object is accessed. Accessing dummy slots from buckets not containing the target object (rather than real slots), on the other hand, is necessary for efficiency: in combination with Ring ORAM's *eviction phase* (discussed next) it lets the stash size remain constant by preventing multiple real objects from being addded to the stash on a single access.

Eviction Phase and Reshuffling. The aforementioned protocol falls short in two ways. First, if objects are placed in the stash after each access, the stash will grow unbounded. Second, all slots will eventually be marked as invalid. Ring

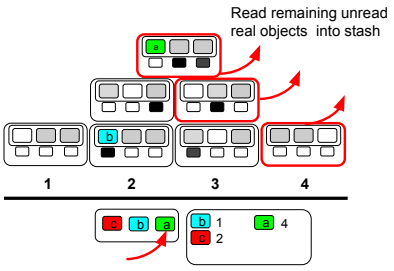


Figure 2: Eviction - Read Phase

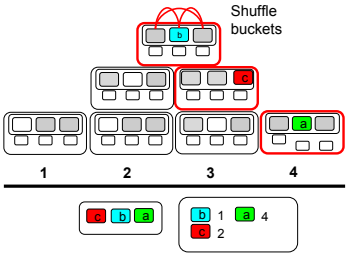


Figure 3: Eviction - Write Phase

ORAM sidesteps these issues through two complementary processes: *eviction* and *bucket reshuffling*. Every A accesses, the *evict path* operation evicts objects from the client stash to cloud storage. It deterministically selects a target path, flushes as much data as possible, and permutes each bucket in the path, revalidating any invalid slots. Evict path consists of a read and write phase. In the read phase, it retrieves Z objects from each bucket in the path: all remaining valid real objects, plus enough valid dummies to reach a total of Z objects read. In the write phase, it places each stashed object—including those read by the read phase—to the deepest bucket on the target path that intersects with the object's assigned path. Evict path then permutes the real and dummy values in each bucket along the target path, marking their slots as *valid*, and writes their contents to server storage. Figure 2 and 3 show the evict path procedure applied to path 4. In the read phase, evict path reads the unread object a from the root node and dummies from other buckets on the path. In the write phase (Fig. 3), a is flushed to leaf 4, as its path intersects completely with the target path. Finally, we note that randomness may cause a bucket to contain only invalid slots before its path is evicted, rendering it effectively unaccessible. When this happens, Ring ORAM restores access to the bucket by performing an *early reshuffle* operation that executes the read phase and write phase of evict path only for the target bucket.

Eviction Security. The read phase leaks no information about the contents of a given bucket. It systematically reads exactly Z valid objects from the bucket, selecting the valid real objects from the z real objects in the bucket, padding the remaining $Z-z$ required reads with a random subset of the S dummy blocks. The random permutation and randomized encryption ensure that the server learns no information about how many real objects exist, and how many have been accessed. Similarly, the write phase hides the values and locations of objects written. At every bucket, the storage server observes only a newly encrypted and permuted set of objects, eliminating any correlation between past and future accesses to that bucket. Together, the read and write phases ensure that no slot is accessed more than once between reshuffles, guaranteeing the bucket invariant.

Similarly, the eviction process leaks no information about the paths of the newly evicted objects: since all paths intersect at the root and the server cannot infer the contents of any individual bucket, any object in the stash may be flushed during *any* evict path. Moreover, since all paths intersect at the root, any object in the stash may be flushed during *any* evict path.

5 System Architecture

Obladi, like most privacy-preserving systems [70, 76, 86] consists of a centralized trusted component, the *proxy*, that communicates with a fault-tolerant but untrusted entity, *cloud storage* (Figure 4). The proxy handles concurrency control, while the untrusted cloud storage stores the private data. Obladi ensures that requests made by the proxy to the cloud storage over the untrusted network do not leak information. We assume that the proxy can crash and that when it does so, its state is lost. This two-tier design allows applications to run a lightweight proxy locally and delegate the complexity of fault-tolerance to cloud storage.

The proxy has two components: (*i*) a *concurrency control unit* and (*ii*) a *data manager* comprised of a *batch manager* and an *ORAM executor*. The batch manager periodically schedules fixed-size batches of client operations that the ORAM executor then executes on a parallel version of Ring ORAM's algorithm. The executor accesses one of two units located on server storage: *the ORAM tree*, which stores the actual data blocks of the ORAM; and *the recovery unit*, which logs all non-deterministic accesses to the ORAM to a write-ahead log [51] to enable secure failure recovery (§8).

6 Proxy Design

The proxy in Obladi has three goals: guarantee good performance, preserve correctness, and guarantee security. To meet these goals, Obladi designs the proxy around the concept of epochs. The proxy partitions time into a set of fixed-length, non-overlapping epochs. Epochs are the granularity at which Obladi guarantees durability and consistency. Each transaction, upon arriving at the proxy, is assigned to an epoch and clients are notified of whether a transaction has committed only when the epoch ends. Until then, Obladi buffers all updates at the proxy.

This flexibility boosts *performance* in two ways. First, it allows Obladi to implement a multiversioned concurrency control (MVCC) algorithm on top of a single versioned Ring ORAM. MVCC algorithms can significantly improve throughput by allowing read operations to proceed with

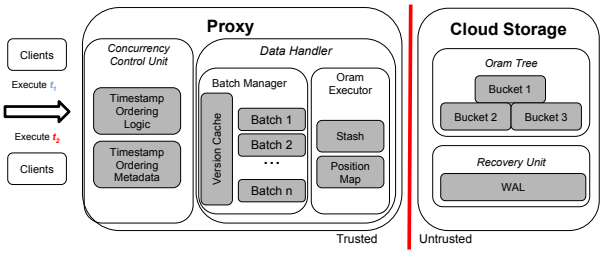


Figure 4: System Architecture

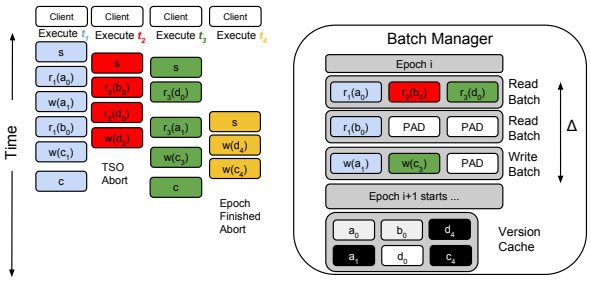


Figure 5: Batching Logic - $r_x(a_y)$ denotes that transaction t_x reads the version of object a written by transaction t_y

limited blocking. These performance gains are especially significant in the presence of long-running transactions or high storage access latency, as is often the case for cloud storage systems. Second, it reduces traffic to the ORAM, as only the database state at the end of the epoch needs to be written out to cloud storage.

Importantly, Obladi's choice to enforce consistency and durability only at epoch boundaries does not affect *correctness*; transactions continue to observe a serializable and recoverable schedule (i.e., committed transactions do not see writes from aborted transactions).

For transactions executing concurrently within the same epoch, serializability is guaranteed by concurrency control; transactions from different epochs are naturally serialized by the order in which the proxy executes their epochs. No transaction can span multiple epochs; unfinished transactions at epoch boundaries are aborted, so that no transaction is ongoing during epoch changes.

Durability is instead achieved by enforcing epoch fate-sharing [82] during proxy or client crashes: Obladi guarantees that either all *completed* transactions (i.e., transactions for which a commit request has been received) in the epoch are made durable or all transactions abort. This way, no *committed* transaction can ever observe non-durable writes.

Finally, the deterministic pattern of execution that epochs impose drastistically simplifies the task of guaranteeing workload independence: as we describe further below, the frequency and timing at which requests are sent to untrusted storage are fixed and consequently independent of the workload.

The proxy processes epochs with two modules: the concurrency control unit (CCU) ensures that execution remains serializable, while the data handler (DH) accesses the actual data objects. We describe each in turn.

6.1 Concurrency Control

Obladi, like many existing commercial databases [57, 65], uses multiversioned concurrency control [10]. Obladi specifically chooses multiversioned timestamp ordering (MVTSO) [10, 68] because it allows uncommitted writes to be immediately visible to concurrently executing transactions. To ensure serializability, transactions log the set of transactions whose uncommitted values they have observed (their write-read dependencies) and abort if any of their dependencies fail to commit. This optimistic approach is critical to Obladi's performance: it allows transactions within the same epoch to see each other's effects even as Obladi delays commits until the epoch ends. In contrast, a pessimistic protocol like two-phase locking [26], which precludes transactions from observing uncommitted writes, would artificially increase contention by holding exclusive write-locks for the duration of an epoch. When a transaction starts, MVTSO assigns it a unique timestamp that determines its serialization order. A write operation creates a new object version marked with its transaction's timestamp and inserts it in the *version chain* associated with that object. A read operation returns the object's latest version with a timestamp smaller than its transaction's timestamp. Read operations further update a *read marker* on the object's version chain with their transaction's timestamp. Any write operation with a smaller timestamp that subsequently tries to write to this object is aborted, ensuring that no read operation ever fails to observe a write from a transaction that should have preceded it in the serialization order.

Consider for example the set of transactions executing in Figure 5. Transaction t_1's update to object a ($w(a_1)$) is immediately observed by transaction t_3 ($r_3(a_1)$). t_3 becomes dependent on t_1 and can only commit once t_1 also commits. In contrast, t_2's write to object d causes t_2 to abort: a transaction with a higher timestamp (t_3) had already read version d_0, setting the version's read marker to 3.

6.2 Data Handler

Once a version is selected for reading or writing, the DH becomes responsible for accessing or modifying the actual object. Whereas it suffices to guarantee durability and consistency only at epoch boundaries, security must hold at all times, posing two key challenges. First, the number of requests executed in parallel can leak information, e.g., data dependencies within the same transaction [11, 70]. Second, transactions may abort (§6.1), requiring their effects to be rolled back without revealing the existence of contended objects [5, 72]. To decouple the demands of these workloads from the timing and set of requests that it forwards to cloud storage, Obladi leverages the following observation: transactions can always be re-organized so that all reads from cloud storage execute before all writes [19, 38, 47, 88]. Indeed, while operations within a transaction may depend on the data

returned by a read from cloud storage, no operation depends on the execution of a write. Accordingly, Obladi organizes the DH into a read phase and a write phase: it first reads all necessary objects from cloud storage, before applying all writes.

Read Phase. Obladi splits each epoch's read phase into a *fixed* set of *R fixed-sized* read batches (b_{read}) that are forwarded to the ORAM executor at *fixed* intervals (Δ_{epoch}). This deterministic structure allows Obladi to execute dependent read operations without revealing the internal control flow of the epoch's transactions. Read operations are assigned to the epoch's next unfilled read batch. If no such batch exists, the transaction is aborted. Conversely, before a batch is forwarded to the ORAM executor, all remaining empty slots are padded with dummy requests. Obladi further *deduplicates* read operations that access the same key. As we describe in §7, this step is necessary for security since parallelized batches may leak information unless requests all access distinct keys [12, 86]. Deduplicating requests also benefits performance by increasing the number of operations that can be served within a fixed-size batch.

Write Phase. While transactions execute, Obladi buffers their write operations into a *version cache* that maintains all object versions created by transactions in the epoch. At the end of an epoch, transactions that have yet to finish executing (recall that epochs terminate at fixed intervals) are aborted and their operations are removed. The latest versions of each object in the version cache according to the version chain are then aggregated in a fixed-size *write batch* (b_{write}) that is forwarded to the ORAM executor, with additional padding if necessary.

This entire process, including write buffering and deduplication, must not violate serializability. The DH guarantees that write buffering respects serializability by directly serving reads from the version cache for objects modified in the current epoch. It guarantees serializability in the presence of duplicate requests by only including the last write of the version chain in a write batch. Since Obladi's epoch-based design guarantees that transactions from a later epoch are serialized after all transactions from an earlier epoch, intermediate object versions can be safely discarded. In this context, MVTSO's requirement that transactions observe the *latest* committed write in the serialization order reduces to transactions reading the tail of the previous epoch's version chain.

In the presence of failures, Obladi guarantees serializability and recoverability by enforcing epoch fate sharing: either all transactions in an epoch are made durable or none are. If a failure arises during epoch e_i, the system simply recovers to epoch e_{i-1}, aborting all transactions in epoch e_i. Once again, this flexibility arises from Obladi delaying commit notifications until epoch boundaries.

Example Execution. We illustrate the batching logic once again with the help of Figure 5. Transactions t_1, t_2, t_3 first execute read operations. These operations are aggregated into the first read batch of epoch i. The values returned by these reads are then *cached* into the version cache. t_2 then executes a write operation, which Obladi also buffers into the version cache. When executing $r_2(d_0)$), t_3 reads object d directly from the version cache (we discuss the security of this step in the next section). Similarly, $r_1(a_1)$ reads the buffered uncommitted version of a. In contrast, Obladi schedules $r_1(b_0)$ to execute as part of the next read batch as b_0 is not present in the version cache. The read batch is then padded to its fixed b_{read} size and executed. t_4 contains no read operations: its write operations are simply executed and buffered at the version cache. Obladi then finalizes the epoch by aborting all transactions (and their dependencies) that have not yet finished executing: t_4 is consequently aborted. Finally, Obladi aggregates the last version of every update into the write batch (skipping version c_1 of object c for instance, instead only writing c_2), before notifying clients of the commit decision.

6.3 Reducing Work

Obladi reduces work in two additional ways: it caches reads within an epoch and allows Ring ORAM to execute write operations without also executing dummy queries. While these optimizations may appear straightforward, ensuring that they maintain workload independence requires care.

Caching Reads. Ring ORAM maintains a client-side stash (§4) that stores ORAM blocks until their eviction to cloud storage. Importantly, a request for a block present in the stash still triggers a dummy request: a dummy object is still retrieved from each bucket along its path. While this access may appear redundant at first, it is in fact necessary to preserve *workload independence*: removing it removes the guarantee that the set of paths that Obladi requests from cloud storage is uniformly distributed. In particular, blocks present in the stash are more likely to be mapped to paths farther away from the one visited by the last evict path, as they correspond to paths that could not be flushed: buckets have limited space for real blocks and blocks mapped to paths that only intersect near the top of the tree are less likely to find a free slot to which they can be flushed. The degree to which this effect skews the distribution leaks information about the stash size, and, consequently, about the workload. To illustrate, consider the execution in Figure 6. Objects mapped to paths 1 and 2 (a, b, and f) were not flushed from the stash in the previous eviction of path 4. When these objects are subsequently accessed, naively reading them from the stash without performing dummy reads skews the set of paths accessed toward the right subtree (paths 3 and 4)

Obladi securely mitigates some of this work by drawing a novel distinction between objects that are in the stash as a result of a logical access and those present because they could not be evicted. The former can be safely accessed without performing a dummy read, while the latter cannot. Objects present in the stash following a logical access are mapped to independently uniformly distributed paths. Ring ORAM's path invariant ensures that, without caching, the

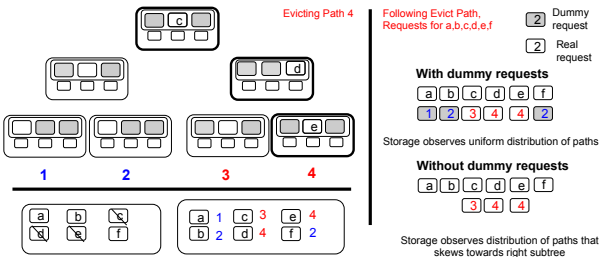


Figure 6: Skew introduced by caching arbitrary objects

set of accessed paths is uniformly distributed. Removing an independent uniform subset of those paths (namely, the dummy requests) will consequently not change the distribution. Thus, caching these objects, and filling out a read batch with other real or dummy requests, preserves the uniform distribution of paths and leaks no information. Obladi consequently allows all read objects to be placed in the version cache for the duration of the epoch. Objects *a*, *b*, *d* are, for instance, placed in the version cache in Figure 5, allowing read $r_2(d_0)$ to read *d* directly from the cache. In contrast, objects present in the stash because they could not be evicted are mapped to paths that skew away from the latest evict path. Caching these objects would consequently skew the distribution of requests sent to the storage away from a uniform distribution, as illustrated in Figure 6.

Dummiless Writes. Ring ORAM must hide whether requests correspond to read or write operations, as the specific pattern in which these operations are interleaved can leak information [89]; that is why Ring ORAM executes a read operation on the ORAM for every access. In contrast, since transactions can always perform all reads before all writes, no information is leaked by informing the storage server that each epoch consists of a fixed-size sequence of potentially dummy reads followed by a fixed-size sequence of potentially dummy writes. Obladi thus modifies Ring ORAM's algorithm to directly place the new version of an object in the stash, without executing the corresponding read. Note, though, that Obladi continues to increment the evict path count on write operations, a necessary step to preserve the bounds on the stash size, which is important for durability (§8).

6.4 Configuring Obladi

Obladi's good performance hinges on appropriately configuring the size/frequency of batches and ORAM tree for a target application. Table 1 summarizes the parameter space.

Ring ORAM. Configuring Ring ORAM first requires choosing an appropriate *Z* parameter. Larger values of *Z* reduce the total size of the ORAM on cloud-storage by decreasing the required height of the ORAM tree and decrease eviction frequency (reducing network/CPU overhead). In contrast, this increase the maximum stash size. Traditional ORAMs thus choose the largest value of Z for which the stash size fits on the proxy. Obladi adds an additional consideration: for durability (as we describe

N	Number of real objects
Z	Number of real slots
S	Number of dummy slots
A	Frequency of evict path
L	Number of levels in the ORAM tree
R	Number of read batches
b_{read}	Size of a read batch
b_{write}	Size of a write batch
Δ	Batch frequency

Table 1: Obladi's configuration parameters

in §8), the stash must be synchronously written out every epoch. One must thus take into account the throughput loss associated with the stash writeback time. Given an appropriate value of Z, Obladi then chooses L, S, and A according to the analytical model proposed in [69].

Epochs and batching. Identifiying the appropriate size and number of batches hinges on several considerations. First, Obladi must provision sufficiently many read batches (R) to handle control flow dependencies within transactions. A transaction that executes in sequence five dependent read operations, will for instance require five read batches to execute (it will otherwise repeatedly abort). Second, the ratio of reads ($R*b_{read}$) to writes (w_{write}) must closely approximate the application's read/write ratio. An overly large write batch will waste resources as it will be padded with many dummy requests. A write batch that is too small will lead to frequent aborts caused by the batch filling up. Third, the size of a read or write batch (respectively b_{read} and b_{write}) defines the degree of parallelism that can be extracted. The desired batch size is thus a function of the concurrent load of the system, but also of hardware considerations, as increasing parallelism beyond an I/O or CPU bottleneck serves no purpose. Finally, the number and frequency of read batches within an epoch increases overall latency, but reduces amortized resource costs through caching and operation pipelining (introduced in §7). Latency-sensitive applications may favor smaller batch sizes, while others may prefer longer epochs, but lower overheads.

Security Considerations. Obladi does not attempt to hide the size and frequency of batches from the storage server (we formalize this leakage in §9). Carefully tuning the size and frequency of batches to best match a given application may thus leak information about the application itself. An OLTP application, for instance, will likely have larger batch sizes (b_{read}), but fewer read batches (R), as OLTP applications sustain a high concurrent load of fairly short transactions. OLAP applications will prefer small or non-existent write batches (b_{write}), as they are predominantly read-only, but require many read batches to support the complex joins/aggregates that they implement. Obladi does not attempt to hide the type of application that is being run. It does, however, continue to hide what data is being accessed and what transactions are currently being run at any given point in time. While Obladi's configuration parameters may, for instance, suggest

that a medical application like FreeHealth is being run, they do not in any way leak information about how, when, or which patient records are being accessed.

7 Parallelizing the ORAM

Existing ORAM constructions make limited use of parallelism. Some allow requests to execute concurrently between eviction or shuffle phases [12, 70, 86], while others target intra-request parallelism to speed up execution of a single request [44]. Obladi explicitly targets both forms of parallelism. Parallelizing Ring ORAM presents three challenges: (*i*) preserving the correct abstraction of a sequential datastore, (*ii*) enforcing security by concealing the position of real blocks in the ORAM (thereby maintaining workload independence), and (*iii*) preserving existing bounds on the stash size. While these issues also arise in prior work [70], the idiosyncrasies of Ring ORAM add new dimensions to these challenges.

Correctness. Obladi makes two observations. First, while all operations conflict at the Ring ORAM tree's root, they can be split into suboperations that access mostly disjoint *buckets* (§4). Second, conflicting bucket operations can be further parallelized by distinguishing accesses to the bucket's metadata from those to its physical data blocks.

Obladi draws from the theory of multilevel serializability [84], which guarantees that an execution is serializable if the system enforces level-by-level serializability: if operation o is ordered before o' at level i, all suboperations of o must precede conflicting suboperations of o'. Thus, if Obladi orders conflicting operations at a level i, it enforces the same order at level $i+1$ for all their conflicting suboperations; conversely, if two operations do not conflict at level i, Obladi executes their suboperations in parallel. To this end, Obladi simply tracks dependencies across operations and orders conflicting suboperations accordingly. Obladi extracts further parallelism in two ways. First, since in Ring ORAM reads to the same bucket between consecutive eviction or reshuffling operations always target different physical data blocks (even when bucket operations conflict on metadata access), Obladi executes them in parallel. Second, Obladi's own batching logic ensures that requests within a batch touch different objects, preventing read and write methods from ever conflicting. Together, these techniques allow Obladi to execute most requests and evictions in parallel.

We illustrate the dependency tracking logic in Figure 7. The read operation to path 1 conflicts with the evict path for path 2, but only at the root (bucket 1). Thus, reads to buckets 2 and 3 can proceed concurrently, even though accesses to the root's metadata must be serialized, as both operations update the bucket access counter and valid/invalid map (§4).

Security. For security, Obladi's parallel evict path operation must flush the same blocks flushed by a sequential implementation. Reproducing this behavior without sacrificing parallelism is challenging. It requires that all real objects

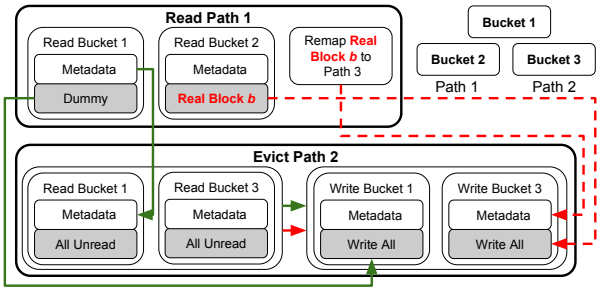


Figure 7: Multilevel Pipelining for a read of path 1 and an evict path of path 2 executing in parallel. Solid green lines represent physical dependencies and dashed red lines represent data dependencies. Inner blocks represent nested operations

brought in during the last A accesses be present in the stash when data is flushed, which may introduce *data dependencies*. Unlike dependencies that arise between operations that access the same physical location in cloud storage, these dependencies are not a deterministic function of an epoch's operations already known to the adversary.

Consider, for instance, block b in Figure 7. In a sequential implementation, b would enter the stash as a result of reading path 1 and be flushed to bucket 3 by the following evict path. Thus, evict path would have to *wait* until b is placed in the stash. Honoring these dependencies opens a timing channel: delay in flushing certain blocks can reveal object placement. As blocks holding real objects can exist anywhere in the tree and be remapped to any path, it follows that it is never secure to execute an eviction operation until all previous access operations have terminated.

Obladi mitigates this restriction by again leveraging delayed visibility and the idea to separate read and write operations within an epoch—but with an important difference. In §6.2 the proxy created separate batches for *logical* read and write operations; to improve parallelism, Obladi, expanding on an idea used by Shroud [44], assigns to separate phases within an epoch the *physical* read and write operations that underlie each of those logical operations. The read phase computes all necessary metadata and executes the set of physical read operations for all logical read path, early reshuffle, and evict path operations. This set is workload independent, so its operations need not be delayed. Physical writes, however, are only flushed at the end of an epoch. The proxy can again apply write deduplication: if a bucket is repeatedly modified during an epoch, only the last version must be written back. Reads that should have read an intermediate write are served locally from the buffered buckets.

The adversary thus always observes a set of reads to random paths followed by a deterministic set of writes independent of the contents of the ORAM and, consequently, of the workload. Data dependencies between read and evict operations no longer create a timing channel. Meanwhile parallelism remains high, as the physical blocks accessed in each phase are guaranteed to be distinct—Ring ORAM directly guarantees this for reads, while bucket deduplication does it for writes.

8 Durability

Obladi guarantees durability at the granularity of epochs: after a crash, it recovers to the state of the last failure-free epoch. Obladi adds two demands to the need of recovering to a consistent state: recovery should leak no information about past or future transactions, and it should be efficient, accessing minimal data from cloud storage. Obladi guarantees the former by ensuring that recovery logic and data logged for recovery maintain workload independence (§3). It strives towards the latter by leveraging the determinism of Ring ORAM.

Consistency. Obladi recovery logic relies on two well-known techniques: write-ahead logging [51] and shadow paging [30]. Obladi mandates that transactions be durable only at the end of an epoch; thus, on a proxy failure, all ongoing transactions can be aborted, and the system reverted to the previous epoch. To make this possible, Obladi must (*i*) recover the proxy metadata lost during the proxy crash, and (*ii*) ensure that the ORAM does not contain any of the aborted transactions' updates. To recover the metadata, Obladi logs three data structures before declaring the epoch committed: the position map, the permutation map, and the stash. The position map and the permutation map identify the position of real objects in the ORAM tree (respectively, in a path and in a bucket); logging them prevents the recovery logic from having to scan the full ORAM to recover the position of buckets. Logging the stash is necessary for correctness. As eviction may be unable to flush the entire stash, some newly written buckets may be present only in the stash, even at epoch boundaries. Failing to log the stash could thus lead to data loss.

To undo partially executed transactions, Obladi adapts the traditional copy-on-write technique of shadow paging [30]: rather than updating buckets in place, it creates new versions of each bucket on every write. Obladi then leverages the inherent determinism of Ring ORAM to reconstruct a consistent snapshot of the ORAM at a given epoch. In Ring ORAM, the current version of a bucket (i.e. the number of times a bucket has been written) is a deterministic function of the number of prior evict paths. The number of evict paths per epoch is similarly fixed (evict paths happen every A accesses, and epochs are of fixed size). Obladi can then trivially revert the ORAM on failures by setting the evict path counter to its value at the end of the last committed epoch. This counter determines the number of evict paths that have occurred, and consequently the object versions of the corresponding epoch.

Security. Obladi ensures that (*i*) the information logged for durability remains independent of data accesses, and (*ii*) that the interactions between the failed epoch, the recovery logic, and the next epoch preserve workload independence.

Obladi addresses the first issue by encrypting the position map and the contents of the permutations table. It similarly encrypts the stash, but also *pads* it to its maximum size, as determined in canonical Ring ORAM [69], to prevent it from indicating skew (if a small number of objects are accessed frequently, the stash will tend to be smaller).

The second concern requires more care: workload independence must hold before, during, and after failures. Ring ORAM guarantees workload independence through two invariants: the bucket invariant and the path invariant (§4). Preserving bucket slots from being read twice between evictions is straightforward. Obladi simply logs the invalid/valid map to track which slots have already been read and recovers it during recovery; there is no need for encryption, as the set of slots read is public information. Ensuring that the ORAM continues to observe a uniformly distributed set of paths is instead more challenging. Specifically, read requests from partially executed transactions can potentially leak information, even when recovering to the previous epoch. Traditionally, databases simply *undo* partially executed transactions, mark them as aborted, and proceed as if they had never existed. From a security standpoint, however, these transactions were still observed by the adversary, and thus may leak information. Consider a transaction accessing object a (mapped to path 1) that aborts because of a proxy failure. Upon recovery, it is likely that a client will attempt to access a again. As the recovery logic restores the position map of the previous epoch, that new operation on a will result in another access to path 1, revealing that the initial access to path 1 was likely real (rather than padded), as the probability of collisions between two uniformly chosen paths is low. To mitigate this concern while allowing clients to request the same objects after failure, Obladi durably logs the list of paths and slot indices that it accesses, before executing the actual requests, and replays those paths during recovery (remapping any real blocks). While this process is similar to traditional database redo logging [51], the goal is different. Obladi does not try to reapply transactions (they have all aborted), but instead forces the recovery logic to be deterministic: the adversary always sees the paths from the aborted epoch repeated after a failure.

Optimizations. To minimize the overhead of checkpointing, Obladi checkpoints deltas of the position, permutation, and valid/invalid map, and only periodically checkpoints the full data structures. While the number of changes to the permutation and valid/invalid maps directly follows from the set of physical requests made to cloud storage, the size of the delta for the position map reveals how many real requests were included in an epoch—padded requests do not lead to position map updates. Obladi thus pads the map delta to the maximum number of entries that could have changed in an epoch (i.e., the read batch size times the number of read batches, plus the size of the single write batch).

9 System Security

We now outline Obladi's security guarantees, deferring a formal treatment to the associated technical report [20]. To the best of our knowledge, we are the first to formalize the notion of crashes in the context of oblivious RAM.

Model We express our security proof within the Universal Composability (UC) framework [14], as it aligns well with

the needs of modern distributed systems: a UC-secure system remains UC-secure under concurrency or if composed with other UC-secure systems. Intuitively, proving security in the UC model proceeds as follows. First, we specify an *ideal functionality* $\mathcal{F}$ that defines the expected functionality of the protocol for both correctness and security. For instance, Obladi requires that the execution be serializable, and that only the frequency of read and write batches be learned. We must ensure that the real protocol provides the same functionality to honest parties while leaking no more information than $\mathcal{F}$ would. To establish this, we consider two different worlds: one where the real protocol interacts with an adversary $\mathcal{A}$, and one where $\mathcal{F}$ interacts with $\mathcal{S}_\mathcal{A}$, our best attempt at simulating $\mathcal{A}$. $\mathcal{A}$'s transcript—including its inputs, outputs, and randomness—and $\mathcal{S}_\mathcal{A}$'s output are given to an environment $\mathcal{E}$, which can also observe all communications within each world. $\mathcal{E}$'s goal is to determine which world contains the real protocol. To prompt the worlds to diverge, $\mathcal{E}$ can delay and reorder messages, and even control external inputs (potentially causing failures). Intuitively, $\mathcal{E}$ represents anything external to the protocol, such as concurrently executing systems. We say that the real protocol is secure if, for any adversary $\mathcal{A}$, we can construct $\mathcal{S}_\mathcal{A}$ such that $\mathcal{E}$ can never distinguish between the worlds.

Assumptions The security of Obladi relies on four assumptions. (*i*) Canonical Ring ORAM is linearizable (*ii*) MVTSO generates serializable executions. (*iii*) The network will retransmit dropped packets. The adversary learns of the retransmissions, but nothing more.

Ideal Functionality To define the ideal functionality $\mathcal{F}_{Ob}$, recall that the proxy is considered trusted while interactions with the cloud storage are not. This allows $\mathcal{F}_{Ob}$ to replace the proxy and intermediate between clients and the storage server, performing the same functions as the proxy (we do not try to hide the concurrency/batching logic). We must, however, define $\mathcal{F}_{Ob}$ to obliviously hide data values and access patterns. To this end, when the proxy logic finalizes a batch, $\mathcal{F}_{Ob}$ simply informs the storage server that it is executing a read or write batch. Since $\mathcal{F}_{Ob}$ is a theoretical ideal, we allow it to manage all storage internally, so it then updates its local storage and furnishes the appropriate response to each client.

In this setup, modeling proxy crashes is straightforward. Crashes can occur at any time and cause the proxy to lose all state. So, on an external input to crash, $\mathcal{F}_{Ob}$ simply clears its state. Since we accept that $\mathcal{A}$ may learn of proxy crashes, $\mathcal{F}_{Ob}$ also sends a message to the storage server that it has crashed.

Proof Sketch The correctness of the system is straightforward, as $\mathcal{F}_{Ob}$ behaves much the same as the proxy.

To prove security, we must demonstrate that, for any algorithm $\mathcal{A}$ defining the behavior of the storage server, we can accurately simulate $\mathcal{A}$'s behavior using only the information provided by $\mathcal{F}_{Ob}$. Note that the simulator $\mathcal{S}_\mathcal{A}$ can run $\mathcal{A}$ internally, as $\mathcal{A}$ is simply an algorithm. Thus we can define $\mathcal{S}_\mathcal{A}$ to operate as follows. When $\mathcal{S}_\mathcal{A}$ receives

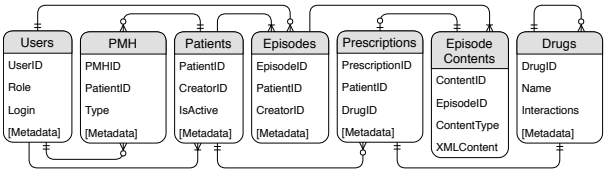


Figure 8: FreeHealth Database Architecture

notification of a batch, it constructs a parallel ORAM batch from uniformly random accesses of the correct type. It provides these accesses to $\mathcal{A}$ and produces $\mathcal{A}$'s response.

The security of this simulation hinges on two key properties: (*i*) the caching and deduplication logic do not affect the distribution of physical accesses, and (*ii*) the physical access pattern of a parallelized batch is entirely determined by the physical accesses proscribed by sequential Ring ORAM for the same batch. The first follows from Ring ORAM's guarantee that each access will be an independent uniformly random path—removing an independently-sampled element does not change the distribution of the remaining set. The second follows from the parallelization procedure simply aggregating all accesses and performing all reads followed by all writes.

These properties ensure that the random access pattern produced by $\mathcal{S}_\mathcal{A}$ is identical to the access pattern produced by the proxy when operating on real data. Thus the simulated $\mathcal{A}$ must behave exactly as it would when provided with real data, and produce indistinguishable output.

10 Implementation

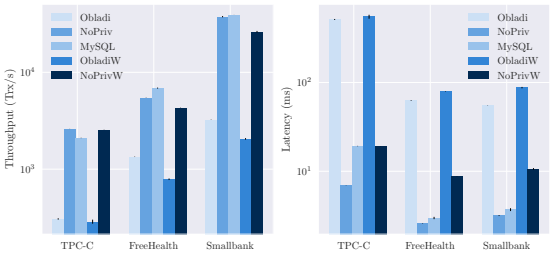
Our prototype consists of 41,000 lines of Java code. We use the Netty library for network communication (v4.1.20), Google protobuffers for serialization (v3.5.1), the Bouncy Castle library (v1.59) for encryption, and the Java MapDB library (v3) for persistence. We additionally implement a non-private baseline (NoPriv). NoPriv shares the same concurrency control logic (TSO), but replaces the proxy data handler with non-private remote storage. NoPriv neither batches nor delays operations; it buffers writes at the local proxy until commit, and serves writes locally when possible.

11 Evaluation

Obladi leverages the flexibility of transactional commits to mitigate the overheads of ORAM. To quantify the benefits and limitations of this approach, we ask:

1. How much does Obladi pay for privacy? (§11.1)
2. How do epochs affect these overheads? (§11.2)
3. Can Obladi recover efficiently from failures? (§11.3)

Experimental Setup The proxy runs on a c5.xlarge Amazon EC2 instance (16 vCPUs, 32GB RAM), and the storage on an m5.4xlarge instance (16 vCPUs, 64GB RAM). The ORAM tree is configured with $Z = 100$ and optimal values of S and A (respectively, 196 and 168) [69]. We report the average of three 90 seconds runs (30 seconds ramp-up/down).



(a) Throughput (b) Latency

Figure 9: Application Performance

Benchmarks We evaluate the performance of our system using three applications: TPC-C [22, 80], SmallBank [22], and FreeHealth [28, 42]. Our microbenchmarks use the YCSB [18] workload generator. **TPC-C**, the defacto standard for OLTP workloads, simulates the business logic of e-commerce suppliers. We configure TPC-C to run with 10 warehouses [87]. In line with prior transactional key-value stores [79], we use a separate table as a secondary index on the `order` table to locate a customer's latest order in the `order status` transaction, and on the `customer` table to look up customers by their last names (`order status` and `payment`). **Smallbank** [22] models a simple banking application supporting money transfers, withdrawals, and deposits. We configure it to run with one million accounts. Finally, we port **FreeHealth** [28, 42], an actively-used cloud EHR system (Figure 8). FreeHealth supports the business logic of medical practices and hospitals. It consists of 21 transaction types that doctors use to create patients and look up medical history, prescriptions, and drug interactions.

11.1 End-to-end Performance

Figure 9 summarizes the results from running the three end-to-end applications in two setups: a local setup in which the latency between proxy and server is low (0.3ms) (**Obladi, NoPriv**), and a more realistic WAN setup with 10ms latency (**ObladiW**, **NoPrivW**). We additionally compare those results with a local MySQL setup. MySQL, unlike NoPriv, cannot buffer writes. We consequently do not evaluate MySQL in the WAN setting.

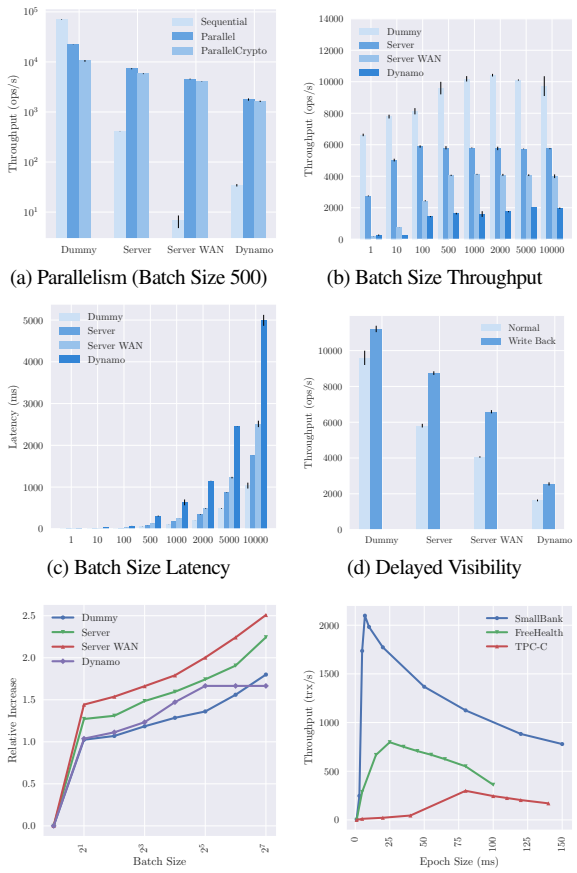
TPC-C Obladi comes within 8× of NoPriv's throughput, as NoPriv is contention-bottlenecked on the high rate of conflicts between the `new-order` and `payment` transactions on the `district` table. NoPriv's performance is itself slightly higher than MySQL as the use of MVTSO allows for the `new-order` and `payment` transactions to be pipelined. In contrast, MySQL acquires exclusive locks for the duration of the transactions. Latency, however, spikes to 70× over NoPriv because of the inflexible execution pattern Obladi needs for security. Transactions in TPC-C vary heavily in size. Epochs must be large enough to accommodate all transactions, and hence artificially increase the latency of short instances. Moreover, write operations must be applied atomically during epoch changes. For a write batch size of 2,000, this process takes on average 340ms, further increasing latency for individual transactions. The write-back process also limits throughput, even preventing non-conflicting operations from making progress (in contrast, NoPriv can benefit from writes never blocking reads in MVTSO). Epoch changes also introduce additional aborts for transactions that straddle epochs. The additional 10ms latency of the WAN setting has comparatively little effect, as the large write batch size of TPC-C is the primary bottleneck: throughput remains within 9x of NoPrivW. Also NoPrivW's performance does not degrade: since MVTSO exposes uncommitted writes immediately, increasing commit latency does not increase contention.

Smallbank Transactions in Smallbank are more homogeneous (between three and six operations); thus, the length of an epoch can be set to more closely approximate most transactions, reducing latency overheads (17× NoPriv). NoPriv is CPU bottlenecked for Smallbank; the relative throughput drop for Obladi is higher (12×) because of the overhead of changing epochs and the blocking that it introduces. Transaction dependency tracking becomes a bottleneck in NoPriv, resulting in a 15% throughput loss over MySQL. Increasing latency between proxy and storage causes both systems' throughput to drop. ObladiW's 35% drop is due to the increased duration of epoch changes (during which no other transactions can execute) while NoPrivW's 30% drop stems from the larger dependency chains that arise from the relatively long commit phase.

FreeHealth Like SmallBank, FreeHealth consists of fairly short transactions and can thus choose a fairly small epoch (five read batches), reducing the impact on latency (20× NoPriv). Unlike Smallbank, however, FreeHealth consists primarily of read operations, and so it can choose a much smaller write batch (200), minimizing the cost of epoch changes and maximizing throughput (only a 4× drop over NoPriv and a 5.5× over NoPrivW for ObladiW). Both NoPriv and Obladi are contention-bottlenecked on the creation of *episodes*, the core units of EHR systems that encapsulate prescriptions, medical history, and patient interaction.

11.2 Impact of Epochs

Though epochs create blocking and cause aborts, they are key to reducing the cost of accessing ORAM, as they allow to (*i*) securely parallelize the ORAM and (*ii*) delay and buffer bucket writes. To quantify epochs' impact on performance as a function of their size and the underlying storage properties, we instantiate an ORAM with 100K objects and choose three different storage backends: a local dummy (storing no real data) that responds to all reads with a static value and ignores writes (`dummy`); a remote server backend with an in-memory hashmap (`server`, ping time 0.3ms) and a remote WAN server backend with an in-memory hashmap (`server WAN`, ping time 10ms); and DynamoDB (`dynamo`, provisioned for 80K req/s, read ping 1ms, write 3ms).



(a) Parallelism (Batch Size 500)
(b) Batch Size Throughput
(c) Batch Size Latency
(d) Delayed Visibility
(e) Epoch Size Impact - ORAM
(f) Epoch Size Impact - Proxy

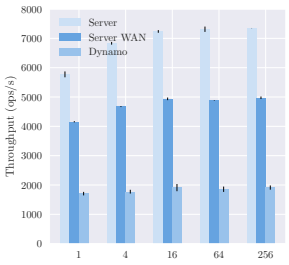
Figure 10: Performance impact of various features

Parallelization We first focus on the performance impact of parallelizing Ring ORAM (ignoring other optimizations). Graph 10a shows that, unsurprisingly, the benefits of parallelism increase with the latency of individual requests. Parallelizing the ORAM for `dummy`, for instance, yields no performance gain; in fact, it results in a 3× slowdown (from 72K req/s to 24K req/s). Sequential Ring ORAM on `dummy` is CPU-bound on metadata computation (remapping paths, shuffling buckets, etc.), so adding coordination mechanisms to guarantee multi-level serializability only increases the cost of accessing a bucket. As storage access latency increases and the ORAM becomes I/O-bound, the benefits of parallelism become more salient. For a batch size of 500, throughput increases by 12× for `server`, as much as 51× for `dynamo`, and 510× for `WAN server`. The available parallelism is a function of both the size/fan-out of the tree and the underlying resource bottlenecks of the proxy. Graph 10b captures the parallelization speedup for both intra- and inter-request parallelism, while Graph 10b quantifies the latency impact of batching. The parallelization speedup achieved for a batch size of one captures intra-request parallelism: the eleven levels of the ORAM can be accessed concurrently, yielding an 11× speedup. As batch sizes increase, Obladi can leverage inter-request parallelism to process non-conflicting physical operations in parallel, with little to no impact on latency. `Dynamo` peaks early (at 1750 req/s) because its client API uses blocking HTTP calls, and `dummy`'s storage eventually bottlenecks on encryption, but `server` and `WAN server` are more interesting. Their throughput is limited by the physical and data dependencies on the upper levels of the tree (recall that paths always conflict at the root (§7)).

Work Reduction To amortize ORAM overheads across a large number of operations, Obladi relies on delayed visibility to buffer bucket writes until the end of an epoch, when they can be executed in parallel, discarding intermediate writes. Reads to those buckets are directly served from the proxy, reducing network communication and CPU work (as encryption is not needed). Graph 10d shows that enabling this optimization for an epoch of eight batches (a setup suitable for FreeHealth and TPC-C) yields a 1.5× speedup on both `dynamo` and the server, a 1.6× speedup on the WAN server, but only minimal gains for `dummy` (1.1×). When using a small number of batches, throughput gains come primarily from combining duplicate operations in buckets near the top of the tree. For example, the root bucket is written 27 times in an epoch of size eight (once per eviction, every 168 requests). As these operations conflict, they must be executed sequentially and quickly become the bottleneck (other buckets have fewer operations to execute). Our optimization lets Obladi write the root bucket only once, significantly reducing latency and thus increasing throughput. As epochs grow in size, increasingly many buckets are buffered locally until the end of the epoch (§7), allowing reads to be served locally and further reducing I/O with the storage. Consider Graph 10e: throughput increases almost logarithmically; metadata computation eventually becomes a bottleneck for `dummy`, while `server` and `server WAN` eventually run out of memory from storing most of the tree (our AWS account did not allow us to provision `dynamo` adequately for larger batches). Larger epochs reduce the raw amount of work per operation: with one batch, Obladi requires 41 physical requests per logical operation, but only requires 24 operations with eight batches. For real transactional workloads, however, epochs are not a silver bullet. Graph 10f suggests that applications are very sensitive to identifying the right epoch duration: too short and transactions cannot make progress, repeatedly aborting; too long and the system will remain unnecessarily idle.

11.3 Durability

Table 11b quantifies the efficiency of failure recovery and the cost it imposes on normal execution for ORAMS of different sizes (we show space results for only the WAN server as Dynamo follows a similar trend). During normal execution, durability imposes a moderate throughput drop (from 0.83× for 10K to 0.89× for 1M). This slowdown is due to the need to checkpoint client metadata and to synchronously log read paths to durable storage before reading. As seen in Graph 11a, computing diffs mitigates the



(a) Checkpoint Frequency (100K)

	10K	100K	1M
Levels	7	11	14
Slowdown	0.83	0.88	0.89
RecTime	1452	2604	6080
Network	182	681	848
Pos	8	74	1610
Perm	15	218	1424
Paths	864	1104	1341

(b) Server Wan Recovery Time (ms)

Figure 11: Durability

impact of checkpointing. Recovery time similarly increases as the ORAM grows, from 1.5s to 6.1s (Table 11b, *RecTime*). The costs of decrypting the position and permutation maps (*Pos* and *Perm*) are low for small datasets, but grow linearly with the number of keys. Read path logging (*Paths*) instead starts much larger, but grows only with the depth of the tree.

12 Related Work

Batching Obladi amortizes ORAM costs by grouping operations into epochs and committing at epoch boundaries. Batching can mitigate expensive security primitives, e.g., it reduces server-side computation in private information retrieval (PIR) schemes [9, 31, 33, 45], amortizes the cost of shuffling networks in Atom [40] and the cost of verifying integrity in Concerto [6]. Changing when operations output commit is a popular performance-boosting technique: it yields significant gains for state-machine replication [35, 37, 64], file systems [55], and transactional databases [21, 47, 82].

ORAM parallelism Obladi extends recent work on parallel ORAM constructions [11, 44, 86] to extract parallelism both *within* and *across* requests. Shroud [44] targets intra-request parallelism by concurrently accessing different levels of tree-based ORAMs. Chung et al [12] and PrivateFS [86] instead target inter-request parallelism, respectively in tree-based [73] and hierarchical [85] ORAMs. Both works execute requests to distinct logical keys concurrently between reshuffles or evictions and deduplicate concurrent requests for the same key to increase parallelism. Obladi leverages delayed visibility to separate batches into read and write phases, extracting concurrency both within requests and across evictions. Furthermore, Obladi parallelizes across requests by deduplicating requests at the trusted proxy.

ObliviStore [77] and Taostore [70] instead approach parallelization by focusing on asynchrony. ObliviStore [77] formalizes the security challenges of scheduling requests asynchronously; the oblivious scheduling mechanism that it presents for that model however is computationally expensive and requires a large stash, making ObliviStore unsuitable for implementing ACID transactions. Like ObliviStore, Taostore leverages asynchrony to parallelize Path ORAM [78], a tree-based construction from which Ring ORAM descends. Taostore, however, targets a different threat model: it assumes both that requests must be processed immediately, and that the timing of responses is visible to the adversary. Request latencies thus necessarily increase linearly with the number of clients [86].

Hiding access patterns for non-transactional systems Many systems seek to provide access pattern protections for analytical queries: Opaque [89] and Cipherbase [5] support oblivious operators for queries that scan or shuffle full tables. Both rely on hardware enclaves for efficiency: Opaque runs a query optimizer in SGX [32], while Cipherbase leverages secure co-processors to evaluate predicates more efficiently. Others seek to hide the parameters of the query rather than the query itself: Olumofin et al. [56] do it via multiple rounds of keyword-based PIR operations [16]; Splinter [83] reduces the number of round-trips necessary by mapping these database queries to function secret sharing primitives. Finally, ObliDB [25] adds support for point queries and efficient updates by designing an oblivious B-tree for indexing. The concurrency control and recovery mechanisms of all these approaches introduce timing channels and structure writes in ways that leak access patterns [5].

Encryption Many commercial systems offer the possibility to store encrypted data [24, 71]. Efficiently executing data-dependent queries like joins, filters, or aggregations without knowledge of the plaintext is challenging: systems like CryptDB [63], Monomi [81], and Seabed [60] tailor encryption schemes to allow executing certain queries directly on encrypted data. Others leverage trusted hardware [7]. In contrast, executing transactions on encrypted data is straightforward: neither concurrency control nor recovery requires knowledge of the plaintext data.

13 Conclusion

This paper presents Obladi, a system that, for the first time, considers the security challenges of providing ACID transactions without revealing access patterns. Obladi guarantees security and durability at moderate cost through a simple observation: transactional guarantees are only required to hold for committed transactions. By delaying commits until the end of epochs, Obladi inches closer to providing practical oblivious ACID transactions.

Acknowledgements We thank our shepherd, Jay Lorch, for his commitment to excellence, and the anonymous reviewers for their helpful comments. We are grateful to Sebastian Angel, Soumya Basu, Vijay Chidambaram, Trinabh Gupta, Paul Grubbs, Malte Schwarzkopf, Yunhao Zhang, and the MIT PDOS reading group for their feedback. This work was supported by NSF grants CSR-1409555 and CNS-1704742, and an AWS EC2 Education Research grant.

References

[1] Agrawal, D., and El Abbadi, A. Locks with Constrained Sharing (Extended Abstract).

[2] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. XPIR: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014. http://eprint.iacr.org/2014/1025.

[3] AMAZON. S3: Simple storage service. https://aws.amazon.com/s3/.

[4] AMAZON. Simple db. https://aws.amazon.com/simpledb/.

[5] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal Security With Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)* (2013).

[6] ARASU, A., EGURO, K., KAUSHIK, R., KOSSMANN, D., MENG, P., PANDEY, V., AND RAMAMURTHY, R. Concerto: A High Concurrency Key-Value Store with Integrity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2017).

[7] BAJAJ, S., AND SION, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2011).

[8] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)* (2011).

[9] BEIMEL, A., ISHAI, Y., AND MALKIN, T. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology (JOFC) 17*, 2 (2004), 125–151.

[10] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion Concurrency Control — Theory and Algorithms. *ACM Trans. Database Syst. 8*, 4 (1983), 465–483.

[11] BINDSCHAEDLER, V., NAVEED, M., PAN, X., WANG, X., AND HUANG, Y. Practicing Oblivious Access on Cloud Storage: The Gap, the Fallacy, and the New Way Forward. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[12] BOYLE, E., CHUNG, K.-M., AND PASS, R. Oblivious Parallel RAM and Applications. In *Theory of Cryptography Conference (TCC)* (2016).

[13] BULCK, J. V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX)* (2018).

[14] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (2001).

[15] CECCHETTI, E., ZHANG, F., JI, Y., KOSBA, A., JUELS, A., AND SHI, E. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[16] CHOR, B., GILBOA, N., AND NAOR, M. Private information retrieval by keywords, 1997.

[17] CLOUD, C. 5 advantages of a cloud-based EHR.

[18] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)* (2010).

[19] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8:1–8:22.

[20] CROOKS, N., BURKE, M., CECCHETTI, E., HAREL, S., AGARWAL, R., AND ALVISI, L. Obladi: Oblivious Serializable Transactions in the Cloud. *CoRR abs/1809.10559* (2018).

[21] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *ACM Symposium on Principles of Distributed Computing (PODC)* (2017).

[22] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases.

[23] DYNAMODB. DynamoDB. https://aws.amazon.com/dynamodb/.

[24] DYNAMODB. Encryption at rest. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html.

[25] ESKANDARIAN, S., AND ZAHARIA, M. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR abs/1710.00458* (2017).

[26] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM 19*, 11 (1976), 624–633.

[27] FLETCHER, C. W., REN, L., KWON, A., AND V. DI, M. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2015).

[28] FREEHEALTH.IO. FreeHealth EHR. https://freehealth.io/. Accessed 2018-05-01.

[29] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[30] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys (CSUR) 13*, 2 (1981), 223–242.

[31] GUPTA, T., CROOKS, N., MULHERN, W., SETTY, S., ALVISI, L., AND WALFISH, M. Scalable and Private Media Consumption with Popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[32] INTEL. Intel Software Guard Extension - SGX. https://software.intel.com/en-us/sgx.

[33] ISHAI, Y., KUSHILEVITZ, E., OSTROVSKY, R., AND SAHAI, A. Batch Codes and Their Applications. In *ACM Symposium on Theory of Computing (STOC)* (2004).

[34] JONES, E. P., ABADI, D. J., AND MADDEN, S. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2010).

[35] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[36] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)* (2019).

[37] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS) 27*, 4 (2010), 7:1–7:39.

[38] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

[39] KUO, A. M.-H. Opportunities and challenges of cloud computing to improve health care services. *Journal of Medical Internet Research (JMIR) 13*, 3 (2011).

[40] KWON, A., CORRIGAN-GIBBS, H., DEVADAS, S., AND FORD, B. Atom: Horizontally Scaling Strong Anonymity. In *ACM Symposium on Operating System Principles (SOSP)* (2017).

[41] LARSON, P.-A., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance Concurrency Control Mechanisms for Main-memory Databases. In *Proceedings of the VLDB Endowment (PVLDB)* (2011).

[42] LIBRE, M. FreeHealth EHR. https://https://freemedsoft.com/fr/. Accessed 2018-05-01.

[43] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX)* (2018).

[44] LORCH, J., PARNO, B., MICKENS, J., RAYKOVA, M., AND SCHIFFMAN, J. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. In *Conference on File and Storage Technologies (FAST)* (2013).

[45] LUEKS, W., AND GOLDBERG, I. Sublinear Scaling for Multi-Client Private Information Retrieval. In *Financial Cryptography and Data Security (FC)* (2015).

[46] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[47] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[48] MICROSOFT. Azure tables. https://azure.microsoft.com/en-us/services/storage/tables/.

[49] MICROSOFT. Documentdb - nosql service for json. https://azure.microsoft.com/en-us/services/documentdb/.

[50] MICROSOFT. SQL Server. https://www.microsoft.com/en-cy/sql-server/sql-server-2016.

[51] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.

[52] MONGODB. Agility, Performance, Scalibility. Pick three. https://www.mongodb.org/.

[53] NARAYANAN, A., AND SHMATIKOV, V. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symposium on Security and Privacy (SP)* (2008).

[54] NARAYANAN, A., AND SHMATIKOV, V. Myths and fallacies of "personally identifiable information". *Commun. ACM 53*, 6 (June 2010), 24–26.

[55] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6:1–6:26.

[56] OLUMOFIN, F., AND GOLDBERG, I. Privacy-preserving Queries over Relational Databases. In *Privacy Enhancing Technologies Symposium (PETS)* (2010).

[57] ORACLE. InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html/.

[58] ORACLE. MySQL. https://www.mysql.com/.

[59] ORACLE. MySQL Cluster. https://www.mysql.com/products/cluster/.

[60] PAPADIMITRIOU, A., BHAGWAN, R., CHANDRAN, N., RAMJEE, R., HAEBERLEN, A., SINGH, H., MODI, A., AND BADRINARAYANAN, S. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[61] PAPADIMITRIOU, C. H. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM) 26*, 4 (1979), 631–653.

[62] PLATFORM, G. C. Cloud spanner. http://cloud.google.com/spanner/.

[63] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM Symposium on Operating System Principles (SOSP)* (2011).

[64] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[65] POSTGRESQL. http://www.postgresql.org/.

[66] REDDY, P. K., AND KITSUREGAWA, M. Speculative Locking Protocols to Improve Performance for Distributed Database Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 16*, 2 (2004), 154–169.

[67] REED, D. P. Implementing Atomic Actions on Decentralized Data (Extended Abstract). In *ACM Symposium on Operating System Principles (SOSP)* (1979).

[68] REED, D. P. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems (TOCS) 1*, 1 (1983), 3–23.

[69] REN, L., FLETCHER, C., KWON, A., STEFANOV, E., SHI, E., VAN DIJK, M., AND DEVADAS, S. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium (USENIX)* (2015).

[70] SAHIN, C., ZAKHARY, V., EL ABBADI, A., LIN, H., AND TESSARO, S. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *IEEE Symposium on Security and Privacy (SP)* (2016).

[71] SERVER, M. S. Always Encrypted. https://www.microsoft.com/en-us/research/project/always-encrypted/.

[72] SHEFF, I., MAGRINO, T., LIU, J., MYERS, A. C., AND VAN RENESSE, R. Safe Serializable Secure Scheduling: Transactions and the Trade-Off Between Security and Consistency. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[73] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011).

[74] SINGEL, R. Netflix spilled your *Brokeback Mountain* secret, lawsuit claims. *Wired* (Dec. 2009). http://www.wired.com/images_blogs/threatlevel/2009/12/doe-v-netflix.pdf.

[75] STEFANOV, E., AND SHI, E. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy (SP)* (2013).

[76] STEFANOV, E., AND SHI, E. ObliviStore: High Performance Oblivious Distributed Cloud Data Store. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[77] STEFANOV, E., SHI, E., AND SONG, D. Towards Practical Oblivious RAM.

[78] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[79] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing Modular Concurrency Control to the Next Level. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2017).

[80] TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page. http://www.tpc.org/tpcc.

[81] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing Analytical Queries over Encrypted Data. In *Proceedings of the VLDB Endowment (PVLDB)* (2013).

[82] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-memory Databases. In *ACM Symposium on Operating System Principles (SOSP)* (2013).

[83] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical Private Queries on Public Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[84] WEIKUM, G. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. Database Syst. 16*, 1 (1991), 132–180.

[85] WILLIAMS, P., SION, R., AND CARBUNAR, B. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[86] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: A Parallel Oblivious File System. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[87] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating System Principles (SOSP)* (2015).

[88] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *ACM Symposium on Operating System Principles (SOSP)* (2015).

[89] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

ASAP: Fast, Approximate Graph Pattern Mining at Scale

Anand Padmanabha Iyer[⋆*] Zaoxing Liu[†*] Xin Jin[†]
Shivaram Venkataraman[•] Vladimir Braverman[†] Ion Stoica[⋆]
[⋆]*UC Berkeley* [†]*Johns Hopkins University* [•]*Microsoft Research / University of Wisconsin*

Abstract

While there has been a tremendous interest in processing data that has an underlying graph structure, existing distributed graph processing systems take several minutes or even hours to mine simple patterns on graphs. This paper presents ASAP, a fast, approximate computation engine for graph pattern mining. ASAP leverages state-of-the-art results in graph approximation theory, and extends it to general graph patterns in distributed settings. To enable the users to navigate the tradeoff between the result accuracy and latency, we propose a novel approach to build the Error-Latency Profile (ELP) for a given computation. We have implemented ASAP on a general-purpose distributed dataflow platform and evaluated it extensively on several graph patterns. Our experimental results show that ASAP outperforms existing exact pattern mining solutions by up to 77×. Further, ASAP can scale to graphs with billions of edges without the need for large clusters.

1 Introduction

The recent past has seen a resurgence in storing and processing massive amounts of graph-structured data [1, 3]. Algorithms for graph processing can broadly be classified into two categories. The first, *graph analysis* algorithms, compute properties of a graph typically using neighborhood information. Examples of such algorithms include PageRank [46], community detection [31] and label propagation [65]. The second, *graph pattern mining* algorithms, discover structural patterns in a graph. Examples of graph pattern mining algorithms include motif finding [44], frequent sub-graph mining (FSM) [60] and clique mining [19]. Graph mining algorithms are used in applications like detecting similarity between graphlets [49] in social networking and for counting pattern frequencies to do credit card fraud detection.

Today, a deluge of graph processing frameworks exist, both in academia and open-source [20, 24, 25, 34–36, 40, 42, 43, 45, 50, 53, 54, 58, 64]. These frameworks typically provide high-level abstractions that make it easy for developers to implement many graph algorithms. A vast majority of the existing graph processing frameworks however have focused on graph analysis algorithms. These frameworks are fast and can scale out to handle very large graph analysis settings: for instance, GraM [59] can run one iteration of page rank on a trillion-edge graph in 140 seconds in a cluster. In contrast, systems that support graph pattern mining fail to scale to even moderately sized graphs, and are slow, taking several hours to mine simple patterns [29, 55].

The main reason for the lack of the scalability in pattern mining is the underlying complexity of these algorithms—mining patterns requires complex computations and storing exponentially large intermediate candidate sets. For example, a graph with a million vertices may possibly contain 10^{17} triangles. While distributed graph-processing solutions are good candidates for processing such massive intermediate data, the need to do expensive joins to create candidates severely degrades performance. To overcome this, Arabesque [55] proposes new abstractions for graph mining in distributed settings that can significantly optimize how intermediate candidates are stored. However, even with these methods, Arabesque takes over 10 hours to *count* motifs in a graph with less than 1 billion edges.

In this paper, we present ASAP[1], a system that enables both *fast* and *scalable* pattern mining. ASAP is motivated by one key observation: *in many pattern mining tasks, it is often not necessary to output the exact answer*. For instance, in FSM the task is to find the *frequency* of subgraphs with an end-goal of ordering them by occurrences. Similarly, motif counting determines the number of occurrences of a given motif. In these scenarios, it is sufficient to provide an *almost* correct answer. Indeed, our conversations with a social network firm [4] revealed that their application for social graph similarity uses a count of similar graphlets [49]. Another company's [4] fraud detection system similarly counts the frequency of pattern occurrences. In both cases, an approximate count is good enough. Furthermore, it is not necessary to materialize *all* occurrences of a pattern[2]. Based on these use cases, we build a system for *approximate* graph pattern mining.

*Equal contribution.

[1] for *A Swift Approximate Pattern-miner*

[2] In fact, it may even be infeasible to output all embeddings of a pattern in a large graph.

Approximate analytics is an area that has gathered attention in big data analytics [6, 13, 32], where the goal is to let the user trade-off accuracy for much faster results. The basic idea in approximation systems is to execute the *exact* algorithm on a small portion of the data, referred to as *samples*, and then rely on the statistical properties of these samples to compose partial results and/or error characteristics. The fundamental assumption underlying these systems is that there exists a relationship between the input size and the accuracy of the results which can be inferred. However, this assumption falls apart when applied to graph pattern mining. In particular, running the exact algorithm on a sampled graph may not result in a reduction of runtime or good estimation of error (§ 2.2).

Instead, in ASAP, we leverage graph approximation theory, which has a rich history of proposing approximation algorithms for mining specific patterns such as triangles. ASAP exploits a key idea that approximate pattern mining can be viewed as equivalent to probabilistically sampling random instances of the pattern. Using this as a foundation, ASAP extends the state-of-the-art probabilistic approximation techniques to *general patterns* in a *distributed* setting. This lets ASAP massively parallelize instance sampling and provide a drastic reduction in runtimes while sacrificing a small amount of accuracy. ASAP captures this technique in a simple API that allows users to plugin code to detect a single instance of the pattern and then automatically orchestrates computation while adjusting the error bounds based on the parallelism.

Further, ASAP makes pattern mining practical by supporting predicate matching and introducing caching techniques. In particular, ASAP allows mining for patterns where edges in the pattern satisfy a user-specified property. To further reduce the computation time, ASAP leverages the fact that in several mining tasks, such as motif finding, it is possible to cache partial patterns that are building blocks for many other patterns. Finally, an important problem in any approximation system is in allowing users to navigate the tradeoff between the result accuracy and latency. For this, ASAP presents a novel approach to build the Error-Latency Profile (ELP) for graph mining: it uses a small sample of the graph to obtain necessary information and applies Chernoff bound analysis to estimate the worst-case error profile for the original graph.

The combination of these techniques allows ASAP to outperform Arabesque [55], a state-of-the-art exact pattern mining solution by up to 77× on the LiveJournal graph while incurring less than 5% error. In addition, ASAP can scale to graphs with billions of edges—for instance, ASAP can count all the 6 patterns in 4-motifs on the Twitter (1.5B edges) and UK graph (3.7B edges) in 22 and 47 minutes, respectively, in a 16 machine cluster.

We make the following contributions in this paper:

- We present ASAP, the first system to our knowledge, that does fast, scalable approximate graph pattern mining on large graphs. (§3)
- We develop a general API that allows users to mine any graph pattern and present techniques to automatically distribute executions on a cluster. (§4)
- We propose techniques that quickly infer the relationship between approximation error and latency, and show that it is accurate across many real-world graphs. (§5)
- We show that ASAP handles graphs with billions of edges, a scale that existing systems failed to reach. (§6)

2 Background & Motivation

We begin by discussing graph pattern mining algorithms and then motivate the need for a new approach to approximate pattern mining. We then describe recent advancements in graph pattern mining theory that we leverage.

2.1 Graph Pattern Mining

Mining patterns in a graph represent an important class of graph processing problems. Here, the objective is to find instances of a given pattern in a graph or graphs. The common way of representing graph data is in the form of a *property graph* [52], where user-defined properties are attached to the vertices and edges of the graph. A *pattern* is an arbitrary subgraph, and pattern mining algorithms aim to output all subgraphs, commonly referred to as *embeddings*, that match the input pattern. Matching is done via sub-graph isomorphism, which is known to be NP-complete. Several varieties of graph pattern mining problems exist, ranging from finding cliques to mining frequent subgraphs. We refer the reader to [7, 55] for an excellent, in-depth overview of graph mining algorithms.

A common approach to implement pattern mining algorithms is to iterate over all possible embeddings in the graph starting with the simplest pattern (e.g., a vertex or an edge). We can then check all *candidate* embeddings, and prune those that cannot be a part of the final answer. The resulting candidates are then expanded by adding one more vertex/edge, and the process is repeated until it is not possible to explore further. The obvious challenge in graph pattern mining, as opposed to graph analysis, is the exponentially large candidate set that needs to be checked.

Distributed graph processing frameworks are built to process large graphs, and thus seem like an ideal candidate for this problem. Unfortunately when applied to graph mining problems, they face several challenges in managing the candidate set. Arabesque [55], a recently proposed distributed graph mining system, discusses these challenges in detail, and proposes solutions to tackle several of them. However, even Arabesque is unable to scale to large graphs due to the need to materialize candidates and exchange them between machines. As an example, Arabesque takes over 10 hours to count motifs of size 3



(a) Uniform edge sampling.

(b) 3-chains in Twitter graph

(c) Triangles in UK graph

Figure 1: Simply extending approximate processing techniques to graph pattern mining does not work.

in a graph with less than a billion edges on a cluster of 20 machines, each having 256GB of memory.

2.2 Approximate Pattern Mining

Approximate processing is an approach that has been used with tremendous success in solving similar problems in both the big data analytics [6, 32] and databases [22, 26, 27], and thus it is natural to explore similar techniques for graph pattern mining. However, simply extending existing approaches to graphs is insufficient.

The common underlying idea in approximate processing systems is to *sample the input* that a query or an algorithm works on. Several techniques for sampling the input exists, for instance, BlinkDB [6] leverages stratified sampling. To estimate the error, approximation systems rely on the assumption that the sample size relates to the error in the output (e.g., if we sample K items from the original input, then the error in aggregate queries, such as `SUM`, is inversely proportional to $\sqrt{K}$). It is straightforward to envision extending this approach to graph pattern mining—given a graph and a pattern to mine in the graph, we first sample the graph, and run the pattern mining algorithm on the sampled graph.

Figure 1a depicts the idea as applied to triangle counting. In this example, the input graph consists of 10 triangles. Using uniform sampling on the edges we obtain a graph with 50% of the edges. We can then apply triangle counting on this sample to get an answer 1. To scale this number to the actual graph, we can use several ways. One naive way is to double it, since we reduced the input by half. To verify the validity of the approach, we evaluated it on the Twitter graph [39] for finding 3-chains and the UK webgraph [17] graph for triangle counting. The relation between the sample size, error and the speedup compared to running on the original graph ($\frac{T_{orig}}{T_{sample}}$) is shown in figs. 1b and 1c respectively.

These results show the fundamental limitations of the approach. We see that there is no relation between the size of the graph (sample) and the error or the speedup. Even very small samples do not provide noticeable speedups, and conversely, even very large samples end up with significant errors. We conclude that the existing approximation approach of *running the exact algorithm on one or more samples of the input is incompatible with graph pattern mining*. Thus, in this paper, we propose a new approach.

2.3 Graph Pattern Mining Theory

Graph theory community has spent significant efforts in studying various approximation techniques for *specific patterns*. The key idea in these approaches is to model the edges in the graph as a *stream* and *sample instances of a pattern* from the edge stream. Then the *probability of sampling* is used to bound the number of occurrences of the pattern. There has been a large body of theoretical work on various algorithms to sample specific patterns and analysis to prove their bounds [8, 11, 21, 38, 47, 48, 56].

While the intuition of using such sampling to approximate pattern counts is straightforward, the technical details and the analysis are quite subtle. Since sampling once results in a large variance in the estimate, multiple rounds are required to bound the variance. Consider triangle counting as an example. Naively, one would design an technique that uniformly samples three edges from the graph without replacement. Since the probability of sampling one edge is $1/m$ in a graph of m edges, the probability of sampling three edges is $1/m^3$. If the sampled three edges form a triangle, we estimate the number of triangles to be m^3 (the expectation); otherwise, the estimation is 0. While such a sampling technique is unbiased, since m is large in practice, the probability that the sampling would find a triangle is very low and the variance of the result is very large. Obtaining an approximated count with high accuracy, would require a large number of trials, which not only consumes time but also memory.

Neighborhood sampling [48] is a recently proposed approach that provides a solution to this problem in the context of a specific graph pattern, triangle counting. The basic idea is to sample one edge and then gradually add more edges until the edges form a triangle or it becomes impossible to form the pattern. This can be analyzed by Bayesian probability [48]. Let's denote E as the event that a pattern is formed, $E_1, E_2, \ldots, E_k$ are the events that edges $e_1, e_2, \ldots, e_k$ are sampled and stored. Thus the probability of a pattern is actually sampled can be calculated as $Pr(E) = Pr(E_1 \cap E_2 \cdots \cap E_k) = Pr(E_1) \times Pr(E_2|E_1) \cdots \times Pr(E_k|E_1, \ldots, E_{k-1})$. Intuitively, compared to the naive sampling, neighborhood sampling

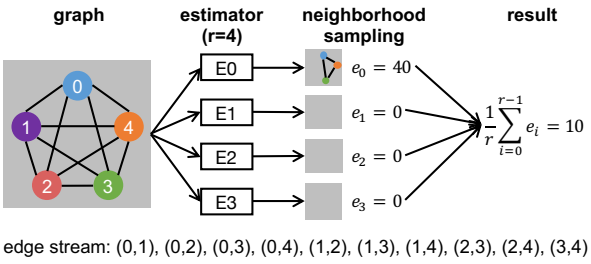


Figure 2: Triangle count by neighborhood sampling

increases the probability that each trial would find an instance of the given pattern, and thus requires fewer estimations to achieve the same accuracy.

2.3.1 Example: Triangle Counting

To illustrate neighborhood sampling, we will revisit the triangle counting example discussed earlier. To sample a triangle from a graph with m edges, we need three edges:

- **First edge l_0.** Uniformly sample one edge from the graph as l_0. The sampling probability $Pr(l_0) = 1/m$.
- **Second edge l_1.** Given that l_0 is already sampled, we uniformly sample one of l_0's adjacent edges (neighbors) from the graph, which we call l_1. Note that neighborhood sampling depends on the ordering of edges in the stream and l_1 appears after l_0 here. The sampling probability $Pr(l_1|l_0) = 1/c$, where c is the number l_0's neighbors appearing after l_0.
- **Third edge l_2.** Find l_2 to finish if edges l_2, l_1, l_0 form a triangle and l_2 appears after l_1 in the stream. If such a triangle is sampled, the sampling probability is $Pr(l_0 \cap l_1 \cap l_2) = Pr(l_0) \times Pr(l_1|l_0) \times Pr(l_2|l_0, l_1) = 1/mc$.

The above technique describes the behaviors of one sampling trial. For each trial, if it successfully samples a triangle, converting probabilities to expectation, $e_i = mc$ will be the estimate of the triangles in the graph. For a total of r trials, $\frac{1}{r}\sum_r e_i$ is output as the approximate result. Figure 2 presents an example of a graph with five nodes.

2.4 Challenges

While the neighborhood sampling algorithm described above has good theoretical properties, there are a number of challenges in building a general system for large-scale approximate graph mining. First, neighborhood sampling was proposed in the context of a specific graph pattern (triangle counting). Therefore, to be of practical use, ASAP needs to generalize neighborhood sampling to other patterns. Second, neighborhood sampling and its analysis assume that the graph is stored in a single machine. ASAP focuses on large-scale, distributed graph processing, and for this it needs to extend neighborhood sampling to computer clusters. Third, neighborhood sampling assumes homogeneous vertices and edges. Real-world graphs are *property graphs*, and in practice pattern mining queries require *predicate matching* which needs the technique to

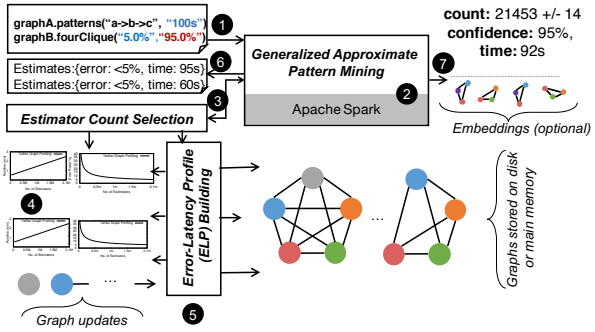


Figure 3: ASAP architecture

be aware of vertex and edge types and properties. Finally, as in any approximate processing system, ASAP needs to allow the end user to trade-off accuracy for latency and hence needs to understand the relation between run-time and error in a distributed setting.

3 ASAP Overview

In this work, we design ASAP, a system that facilitates fast and scalable approximate pattern mining. Figure 3 shows the overall architecture of ASAP. We provide a brief overview of the different components, and how users leverage ASAP to do approximate pattern mining in this section to aid the reader in following the rest of this paper.

User interface. ASAP allows the users to tradeoff accuracy for result latency. Specifically, a user can perform pattern mining tasks using the following two modes ❶:

- **Time budget T.** The user specifies a time budget T, and ASAP returns the most accurate answer within T with a error rate guarantee e and a configurable confidence level (default of 95%).
- **Error budget ϵ.** The user gives an error budget ϵ and confidence level, and ASAP returns an answer within ϵ in the shortest time possible.

Before running the algorithm, ASAP first returns to the user its estimates on the time or error bounds it can achieve ❻. After user approves the estimates, the algorithm is run and the result presented to the user consists of the count, confidence level and the actual run time ❼. Users can also optionally ask to output actual (potentially large number of) embeddings of the pattern found.

Development framework. All pattern mining programs in ASAP are versions of generalized approximate pattern mining ❷ we describe in detail in §4. ASAP provides a standard library of implementations for several common patterns such as triangles, cliques and chains. To allow developers to write program to mine *any* pattern, ASAP further provides a simple API that lets them utilize our approximate mining technique (§ 4.1.2). Using the API, developers simply need to write a program that finds a *single instance* of the pattern they are interested in, which we refer to as *estimator* in the rest of this paper. In a

nutshell, our approximate mining approach depends on running multiple such estimators in parallel.

Error-Latency Profile (ELP). In order to run a user program, ASAP first must find out how many estimators it needs to run for the given bounds ❸. To do this, ASAP builds an ELP. If the ELP is available for a graph, it simply queries the ELP to find the number of estimators ❹. Otherwise, the system builds a new ELP ❺ using a novel technique that is extremely fast and can be done online. We detail our ELP building technique in §5. Since this phase is fast, ASAP can also accommodate graph updates; on large changes, we simply rebuild the ELP.

System runtime. Once ASAP determines the number of estimators necessary to achieve the required error or time bounds, it executes the approximate mining program using a distributed runtime built on Apache Spark [62, 63].

4 Approximate Pattern Mining in ASAP

We now present how ASAP enables large-scale graph pattern mining using neighborhood sampling as a foundation. We first describe our programming abstraction(§ 4.1) that generalizes neighborhood sampling. Then, we describe how ASAP handles errors that arise in distributed processing(§ 4.2). Finally, we show how ASAP can handle queries with predicates on edges or vertices(§ 4.3).

4.1 Extending to General Patterns

To extend the neighborhood sampling technique to general patterns, we leverage one simple observation: at a high level, neighborhood sampling can be viewed as consisting of two phases, *sampling* phase and *closing* phase. In the *sampling* phase, we select an edge in one of two ways by treating the graph as an ordered stream of edges: (a) sample an edge randomly; (b) sample an edge that is adjacent to any previously sampled edges, from the remainder of the stream. In the *closing* phase, we wait for one or more specific edges to complete the pattern.

The probability of sampling a pattern can be computed from these two phases. The closing phase always has a probability of 1 or 0, depending on whether it finds the edges it is waiting for. The probability of the sampling phase depends on how the initial pattern is formed and is a choice made by the developer. For a general graph pattern with multiple nodes, there can be multiple ways to form the pattern. For example, there are two ways to sample a four-clique with different probabilities, as shown in Figure 4. (*i*) In the first case, the sampling phase finds three adjacent edges, and the closing phase waits for rest three edges to come, in order to form the pattern. The sampling probability is $\frac{1}{mc_1c_2}$, where c_1 is the number of the first edge's neighbors and c_2 represents the neighbor count of the first and the second edges. (*ii*) In the second case, the sampling phase finds two disjoint edges, and

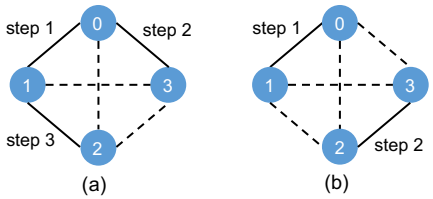


Figure 4: Two ways to sample four cliques. (a) Sample two adjacent edges $(0,1)$ and $(0,3)$, sample another adjacent edge $(1,2)$, and wait for the other three edges. (b) Sample two disjoint edges $(0,1)$ and $(2,3)$, and wait for the other four edges.

the closing phase waits for other four edges to form the pattern. The sampling probability in this case is $\frac{1}{m^2}$.

4.1.1 Analysis of General Patterns

We now show how neighborhood sampling, when captured using the two phases, can extend to general patterns.

Definition 4.1 (General Pattern)**.** *We define a "general pattern" as a set of k connected vertices that form a subgraph in a given graph.*

First, let's consider how an estimator can (possibly) find any general patterns. We show how to sample one general pattern from the graph uniformly with a certain success probability, taking 2 to 5-node patterns as examples. Then, we turn to the problem of maintaining $r \leq 1$ pattern(s) sampled with replacement from the graph. We sample r patterns and a reasonably large r will yield a count estimate with good accuracy. For the convenience of the analysis, we define the following notations: input graph $G = (V, E)$ has m edges and n vertices, and we denote the occurrence of a given pattern in G as $f(G)$. A pattern $p = \{e_i, e_j, \dots\}$ contains a set of ordered edges, i.e., e_i arrives before e_j when $i < j$. When describing the operation of an estimator, $c(e)$ denotes the number of edges adjacent to e and appearing after e, and c_i is $c(e_1, \dots, e_i)$ for any $i \geq 1$. For a given a pattern p^* with k^* vertices, the technique of neighborhood sampling produces p^* with probability $Pr[p = p^*, k = k^*]$. The goal of one estimator is to fix all the vertices that form the pattern, and complete the pattern if possible.

Lemma 4.2. *Let p^* be a k-node pattern in the graph. The probability of detecting the pattern $p = p^*$ depends on k and the different ways to sample using neighborhood sampling technique.*
(1) When $k = 2$, the probability that $p = p^*$ after processing all edges in the graph by all possible neighborhood sampling ways is

$$Pr[p = p^*, k = 2] = \frac{1}{m}$$

(2) When $k = 3$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 3] = \frac{1}{m \cdot c_1}$$

(3) When $k = 4$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 4] = \frac{1}{m^2} \text{ (Type-I) or } \frac{1}{m \cdot c_1 \cdot c_2} \text{ (Type-II)}$$

(4) When $k = 5$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 5] = \frac{1}{m^2 \cdot c_1} \text{ (Type-I)}$$
$$or = \frac{1}{m^2 \cdot c_2} \text{ (Type-II.a)}$$
$$or = \frac{1}{m \cdot c_1 \cdot c_2 \cdot c_3} \text{ (Type-II.b)}$$

Proof. Since a pattern is connected, the operations in the sampling phase are able to reach all nodes in a sampled pattern. To fix such a pattern, the neighborhood sampling needs to confirm all the vertices that form the pattern. Once the vertices are found, the probability of completing such a pattern is fixed.

When $k = 2$, let $p^* = \{e_1\}$ be an edge in the graph. Let $\mathcal{E}_1$ be the event that e_1 is found by neighborhood sampling. There is only one way to fix two vertices of the pattern—uniformly sampling an edge from the graph. By reservoir sampling, we claim that

$$Pr[p = p^*, k = 2] = Pr[\mathcal{E}_1] = \frac{1}{m}$$

When $k = 3$, we need to fix one more vertex beyond the case of $k = 2$. As shown in [48], we need to sample an edge e_2 from e_1's neighbors that occur in the stream after e_1. Let $\mathcal{E}_2$ be the event that e_2 is found. Since $Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{c(e_1)}$,

$$Pr[p = p^*, k = 3] = Pr[\mathcal{E}_1] \cdot Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{m \cdot c(e_1)}$$

When $k = 4$, we require one more step from the case of $k = 2$ or the case of $k = 3$, from extending neighborhood sampling. By extending from the case of $k = 2$ (denoted as Type-I), two more vertices are needed to fix a 4-node pattern. In Type-I, we independently find another edge e_2^* that is not adjacent to the sampled edge e_1. Let $\mathcal{E}_2^*$ be the event that e_2^* is found. Since $Pr[\mathcal{E}_2^*|\mathcal{E}_1] = \frac{1}{m}$,

$$Pr[p = p^*, k = 4] = Pr[p = p^*, k = 2] * Pr[\mathcal{E}_2^*|\mathcal{E}_1]$$
$$= \frac{1}{m^2} \text{ (Type-I)}$$

When extending from the case $k = 2$ (denoted as Type-II), one more vertex is needed to fix a 4-node pattern. In Type-II, we sample a "neighbor" e_3 that comes after $e_1 and e_2$. Let $\mathcal{E}_3$ be the event that e_3 is found. Since e_3 is sampled uniformly from the neighbors of e_1 and e_2 and is appearing after e_1, e_2, $Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{c(e_1,e_2)}$. Thus,

$$Pr[p = p^*, k = 4] = Pr[p = p^*, k = 3] \cdot Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2]$$
$$= \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2)} \text{ (Type-II)}$$

When $k = 5$, we again need one more step from the case $k = 3$ or the case $k = 4$. By extending from $k = 3$ (denoted as Type-I), we require two separate vertices to fix a 5-node pattern. In Type-I, we independently sample another edge e_3^* that is not adjacent to e_1, e_2. Let $\mathcal{E}_3^*$ be the event that e_3^* is found. $Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{m}$. Therefore,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 3] * Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2]$$
$$= \frac{1}{m^2 \cdot c(e_1)} \text{ (Type-I)}$$

When extending from the case $k = 4$, we need to consider the two types separately. By extending Type-I of case $k = 4$ (denoted as Type-II.a), we need one more vertex to construct a 5-node pattern and thus we sample a neighboring edge e_4. Let $\mathcal{E}_4$ be the event that e_4 is found. Since e_4 is sampled from the neighbors of e_1, e_2,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 4] * Pr[\mathcal{E}_4|\mathcal{E}_1, \mathcal{E}_2^*]$$
$$= \frac{1}{m^2 \cdot c(e_1, e_2)} \text{ (Type-II.a)}$$

Similarly, by extending Type-II of case $k = 4$ (denoted as Type-II.b),

$$Pr[p = p^*, k = 5] = \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2) \cdot c(e_1, e_2, e_3)}$$

□

Lemma 4.3. *For pattern p^* with k^* nodes, let's define*

$$\tilde{t} = \begin{cases} \frac{1}{Pr[p=p^*, k=k^*]} & \text{if } p \neq \emptyset \\ 0 & \text{if } p = \emptyset \end{cases}$$

Thus, $E[\tilde{t}] = f(G)$.

Proof. By Lemma 4.2, we know that one estimator samples a particular pattern p^* with probability $Pr[p = p^*, k = k^*]$. Let $p(G)$ be the set of a given pattern in the graph,

$$E[\tilde{t}] = \sum_{p^* \in p(G)} \tilde{t}(p \neq \emptyset) \cdot Pr[p = p^*, k = k^*] = |p(G)| = f(G)$$

□

The estimated count is the average of the input of all estimators. Now, we consider how many estimators are needed to maintain an ϵ error guarantee.

Theorem 4.4. *Let $r \geq 1$, $0 < \epsilon \leq 1$, and $0 < \delta \leq 1$. There is an $O(r)$-space bounded algorithm that return an ϵ-approximation to the count of a k-node pattern, with probability at least $1 - \delta$. For a certain ϵ, when $k = 4$, we need $r \geq \frac{C_1 m^2}{f(G)}$ Type-I estimators, or $r \geq \frac{C_2 m \Delta^2}{f(G)}$ Type-II estimators for some constants C_1 and C_2, to achieve ϵ-approximation in the worst case; When $k = 5$, we need $r \geq \frac{C_3 m^2 \Delta}{f(G)}$ Type-I estimators, or $r \geq \frac{C_4 m^2 \Delta}{f(G)}$ Type-II.a estimators, or $r \geq \frac{C_5 m \Delta^3}{f(G)}$ Type-II.b estimators, for some constants C_3, C_4, C_5 in the worst case.*

API	Description
`sampleVertex`: ()→(v,p)	Uniformly sample one vertex from the graph.
`SampleEdge`: ()→(e,p)	Uniformly sample one edge from the graph.
`ConditionalSampleVertex`: (`subgraph`)→(v,p)	Uniformly sample a vertex that appears after a sampled subgraph.
`ConditionalSampleEdge`: (`subgraph`)→(e,p)	Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.
`ConditionalClose`: (`subgraph`, `subgraph`)→$boolean$	Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.

Table 1: ASAP's Approximate Pattern Mining API.

Proof. Let's first consider the case $k = 4$. Let X_i for $i = 1,\ldots,r$ be the output value of i-th estimator. Let $\bar{X} = \frac{1}{r}\sum_{i=1}^{r} X_i$ be the average of r estimators. By Lemma 4.3, we know that $E[X_i] = f(G)$ and $E[\bar{X}] = f(G)$. From the properties of graph G, we have $c(e) \leq \Delta$ for $\forall e \in E$, where Δ is the maximum degree (note that in practice Δ isn't a tight bound for the edge neighbor information). In Type-I, $X_i \leq m^2$ and we construct random variables $Y_i = \frac{X_i}{m^2}$ such that $Y_i = [0,1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{m^2}$. Thus the probability that the estimated number of patterns has a more than ϵ relative error off its expectation $f(G)$ is $Pr[\bar{X} > (1+\epsilon)f(G)] \leq \frac{\delta}{2}$, which is at most

$$Pr[\sum_{i=1}^{r} Y_i > (1+\epsilon)E[Y]] \leq e^{-\frac{\epsilon^2}{2+\epsilon}E[Y]} \leq e^{-\frac{\epsilon^2}{3}E[Y]} \leq \frac{\delta}{2}$$

by Chernoff bound. Thus $r \geq \frac{3m^2}{\epsilon^2 f(G)} \cdot \ln\frac{2}{\delta}$. Similarly, this lower bound of r holds for $Pr[\bar{X} < (1-\epsilon)f(G)]$.

In Type-II, $X_i \leq 6m\Delta^2$. Let $Y_i = \frac{X_i}{6m\Delta^2}$ such that $Y_i = [0,1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{6m\Delta^2}$. By Chernoff bound, $r \geq \frac{18m\Delta^2}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$. Similarly, when $k = 5$, we (theoretically) need $\frac{6m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-I estimators, $\frac{12m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.a estimators, and $\frac{24m\Delta^3}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.b estimators. Since each estimator stores $O(1)$ edges, the total memory is $O(r)$. □

4.1.2 Programming API

ASAP automates the process of computing the probability of finding a pattern, and derives an expectation from it by providing a simple API that captures two phases. The API, shown in Table 1, consists of the following five functions:

- `SampleVertex` uniformly samples one vertex from the graph. It takes no input, and outputs v and p, where v is the sampled vertex, and p is the probability that sampled v, which is the inverse of the number of vertices.
- `SampleEdge` uniformly samples one edge from the graph. It also takes no input, and outputs e and p, where e is the sampled edge, and p is the sampling probability, which is the inverse of the number of edges of the graph.
- `ConditionalSampleVertex` conditionally samples one vertex from the graph, given *subgraph* as input. It outputs v and p, where v is the sampled vertex and p is the probability to sample v given that *subgraph* is already sampled.
- `ConditionalSampleEdge`(`subgraph`) conditionally samples one edge adjacent to *subgraph* from the graph, given that *subgraph* is already sampled. It outputs e and p, where e is the sampled edge and p is the probability to sample e given *subgraph*.
- `ConditionalClose`(`subgraph`, `subgraph`) waits for edges that appear after the first *subgraph* to form the second *subgraph*. It takes the two subgraphs as input and outputs *yes*/*no*, which is a boolean value indicating whether the second *subgraph* can be formed. This function is usually used as the final step to sample a pattern where all nodes of a possible instance have been fixed (thereby fixing the edges needed to complete that instance of the pattern) and the sampling process only awaits the additional edges to form the pattern.

These five APIs capture the two phases in neighborhood sampling and can be used to develop pattern mining algorithms. To illustrate the use of these APIs, we describe how they can be used to write two representative graph patterns, shown in Figure 5.

Chain. Using our API to write a sampling function for counting three-node chains is straightforward. It only includes two steps. In the first step, we use `SampleEdge` `()` to uniformly sample one edge from the graph (line 1). In the second step, given the first sampled edge, we use `ConditionalSampleEdge` (`subgraph`) to find the second edge of the three-node chain, where `subgraph` is set to be the first sampled edge (line 2). Finally, if the algorithm cannot find e_2 to form a chain with e_1 (line 3), it estimates the number of three-node chains to be 0; otherwise, since the probability to get e_1 and e_2 is $p_1 \cdot p_2$, it estimates the number of chains to be $1/(p_1 \cdot p_2)$.

Four clique. Similarly, we can extend the algorithm of sampling three node chains to sample four cliques. We first sample a three-node chain (line 1-2). Then we sample an adjacent edge of this chain to find the fourth node (line 4). Again, during the three steps, if any edges were not

SampleThreeNodeChain

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
if (!e2)
  return 0
else
  return 1/(p1.p2)
```

SampleFourCliqueType1

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
(e3, p3) = ConditionalSampleEdge(Subgraph(e1, e2))
if (!e3) return 0
subgraph1 = Subgraph(e1,e2,e3)
subgraph2 = FourClique(e1,e2,e3)-subgraph1
if (ConditionalClose(subgraph1,subgraph2))
  return 1/(p1.p2.p3)
else return 0
```

Figure 5: Example approximate pattern mining programs written using ASAP API

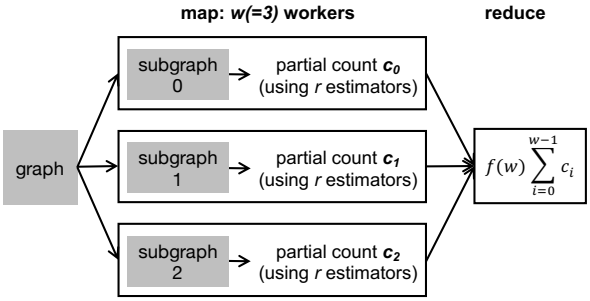


Figure 6: Runtime with graph partition.

sampled, the function would return 0 as no cliques would be found (line 3 and 5). Given e_1, e_2 and e_3, all the four nodes are fixed. Therefore, the function only needs to wait for all edges to form a clique (line 8-9). If the clique is formed, it estimates the number of cliques to be $1/(p_1 \cdot p_2 \cdot p_3)$; otherwise, it returns 0 (line 10). Figure 4(a) illustrates this sampling procedure (CliqueType1).

4.2 Applying to Distributed Settings

Capturing general graph pattern mining using the simple two phase API allows ASAP to extend pattern mining to distributed settings in a seamless fashion. Intuitively, each execution of the user program can be viewed as an instance of the sampling process. To scale this up, ASAP needs to do two things. First, it needs to parallelize the sampling processes, and second, it needs to combine the outputs in a meaningful fashion that preserves the approximation theory.

For parallelizing the pattern mining tasks, ASAP's runtime takes the pattern mining program and wraps it into an *estimator*[3] task. ASAP first partitions the vertices in the graph across machines and executes many copies of the estimator task using standard dataflow operations: *map* and *reduce*. In the map phase, ASAP schedules several copies of the estimator task on each of the machines. Each estimator task operates on the local subgraph in each machine and produces an output, which is a partial count. ASAP's runtime ensures that each estimator in a machine sees the graph's edges and vertices in the *same order*, which is important for the sampling process to produce correct results. Note that although every estimator in each partition sees the graph in the same order, there is *no restriction* on what the order might be (e.g., there is no sorting requirement), thus ASAP uses a random ordering which is fast and requires no pre-processing of the graph. Once this is completed, ASAP runs a reduce task to combine the partial counts and obtain the final answer. This is depicted in fig. 6. This massively parallel execution is one of the reasons for huge latency reduction in ASAP. Since the input to the reduce phase is simply an array of numbers, ASAP's shuffle is extremely light-weight, compared to a system that produces exact answers (and needs to exchange intermediate patterns).

Handling Underestimation. Only summing up the partial counts in the reduce phase underestimates the total number of instances, because when vertices are partitioned to the workers, the instances that span across the partitions are not counted. This results in our technique underestimating the results, and makes the theoretical bounds in neighborhood sampling invalid. Thus, ASAP needs to estimate the error incurred due to distributed execution and incorporate that in the total error analysis.

We use probability theory to do this estimation. We enforce that the vertices in the graph are uniformly randomly distributed across the machines. ASAP is not affected by the normal shortcomings of random vertex partitioning [35] as the amount of data communication is independent of partitioning scheme used. In this case random vertex partitioning is in fact simple to implement, and allows us to theoretically analyze the underestimation.

The theoretical proof for handling the underestimation is outside the scope of this paper. Intuitively, we can think of the random vertex partitioning into w workers as uniform vertex coloring from w available colors. Vertices with the same color are at the same worker and each worker estimates patterns locally on its monochromatic vertices. By doing this coloring, the occurrence of a pattern has been reduced by a factor of $1/f(w)$, where f is a function of the number of workers and the pattern. For instance, a locally sampled triangle has three monochromatic vertices and the probability that this happens among all triangles is $1/w^2$. Thus by the linearity of expectation, each such triangle is scaled by $f(w) = w^2$. A rigorous proof on the maximum possible w with small errors (in practice

[3]Since each program is providing an estimate of the final answer.

w can be >> 100), can be shown using concentration bounds and Hajnal-Szemerédi Theorem [47]. Similarly, each monochromatic 4-clique is scaled by $f(w) = w^3$ and $f(w)$ can be computed for any given pattern.

4.3 Advanced Mining Patterns

Predicate Matching. In property graphs, the edges and vertices contain properties; and thus many real-world mining queries require that matching patterns satisfy some predicates. For example, a *predicate query* might ask for the count of all four cliques on the graph where every vertex in the clique is of a certain type. ASAP supports two types of predicates on the pattern's vertices and edges **all** and **atleast-one**.

For "all" predicate, queries specify a predicate that is applied to *every vertex or edge*. For example, such query may ask for "four cliques where all vertices have a weight of atleast 10". To execute such queries, ASAP introduces a *filtering* phase where the predicate condition is applied before the execution of the pattern mining task. This results in a new graph which consists only of vertices and edges that satisfy the predicate. On this new graph, ASAP runs the pattern mining algorithm. Thus, the "all" predicate query does not require any changes to ASAP's pattern mining algorithm.

The "atleast-one" predicate allows specifying a condition that *atleast* one of the vertices or edges in the pattern satisfies. An example of such a query is "four cliques where atleast one edge has a weight of 10". To execute such predicate queries, we modify the execution to take two passes on the edge list. In the first pass, edges that match the predicate are copied from the `original` edge list to a `matched` edge list. Every entry in the `matched` list is a tuple, `(edge, pos)`, where `pos` is the position in the original list where the matched edge appears. In the second pass, every estimator picks the first edge randomly from the `matched` list. This ensures that the pattern found by the estimator (if it finds one) satisfies the predicate. For the second edge onwards, the estimator uses the `original` list but starts the search from the position at which the first matched edge was found. This ensures that ASAP's probability analysis to estimate the error holds.

Motif mining. Another query used in many real-world workloads is to find all patterns with a certain number of vertices. We define these as *motif queries*; for example a 3-motif query will look for two patterns, triangles and 3-chains. Similarly a 4-motif query looks for six patterns [51]. For motif mining we notice that several patterns have the same underlying *building block*. For example, in 4-motifs, 3-chains are used in many of the constituent patterns. To improve performance, ASAP saves the *sampling* phase's state for the building block pattern. This state includes (i) the currently sampled edges, (ii) the probability of sampling at that point, and (iii) the position in the edge list up to which the estimator has traversed. All the patterns that use this building block are then executed starting from the saved state. This technique can significantly speedup the execution of motif mining queries and we evaluate this in Section 6.2.

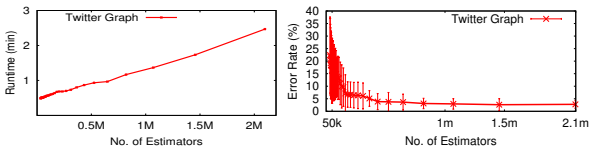


Figure 7: The actual relations between number of estimators and run-time or error rate.

Refining accuracy. In many mining tasks, it is common for the user to first ask for a low accuracy answer, followed by a higher accuracy. For example, users performing exploratory analysis on graph data often would like to iteratively refine the queries. In such settings, ASAP caches the state of the estimator from previous runs. For instance, if a query with an error bound of 10% was executed using 1 million estimators, ASAP saves the output from these estimators. Later, when the same pattern is being queried, but with an error bound of 5% that requires 3 million estimators, ASAP only needs to launch 2 million, and can reuse the first 1 million.

5 Building the Error-Latency Profile (ELP)

A key feature in any approximate processing system is allowing users to trade-off accuracy for result latency. To do this for graph mining, we need to understand the relation between running time and error.

In ASAP's general, distributed graph pattern mining technique described earlier, the only configurable parameter is the number of *estimator* processes used for a mining task. By using r estimators and making r sufficient large, ASAP is able to get results with bounded errors. Since an estimator takes computation and memory resource to sample a pattern, picking the number of estimators r provides a trade-off between result accuracy and resource consumption. In other words, setting a specific number of estimators, N_e, results in a fixed runtime and an error within a certain bound. As an example, fig. 7 depicts the relation between the number of estimators, runtime and error for triangle counting run on the Twitter graph [39]. To enable the user to traverse this trade-off, ASAP needs to determine the correct number of estimators given an error or time budget.

5.1 Building Estimator vs. Time Profile

The time complexity of our approximation algorithm is linearly related to the number of edges in the graph and the number of estimators. Given a graph and a particular pattern, we find the computation time is dominated by the number of estimators when the number of estimators is large enough. From fig. 7, we see that the estimator-time

Algorithm 1 BuildTimeProfile(T^*)

```
1: P ← ∅ // store points for the profile
2: T ← 0, t ← 0, α ← α* // α* can be a reasonable random start
3: while T + t <= T* do
4:     t ← run approximation algorithm with α estimators
5:     P.add((α, t))
6:     α ← 2α
7:     T ← T + t
```

curve is close to linear when the number of estimators is greater than 0.5M. Thus we propose using a linear model to relate the running time to the number of estimators.

When the number of estimators is small, the computation time is also affected by other factorsand thus the curve is not strictly linear. However, for these regions, it is not computationally expensive to profile more exhaustively. Therefore, to build the time profile, we exponentially space our data collection, gathering more points when the number of estimators is small and fewer points as the number of estimators grows. We use a profiling budget T^* to bound the total time spent on profiling. Algorithm 1 shows the pseudo code. ASAP starts from using a small number of estimators ($\alpha \leftarrow \alpha^*$), and doubles α each time until the total profiling time exceeds the profiling cost T^*. In practice, we have found that setting T^* in the minute granularity gives us good results.

5.2 Building Estimator vs. Error Profile

Since error profile is non-linear (fig. 7), techniques like extrapolating from a few data points is not directly applicable. Some recent work has leveraged sophisticated techniques, such as experiment design [57] or Bayesian optimization [12] for the purpose of building non-linear models in the context of instance selection in the cloud. However, these techniques also require the system to compute the error for a given setting for which we need to know the ground-truth, say, by running the exact algorithm on the graph. Not only is this infeasible in many cases, it also undermines the usefulness of an approximation system.

In ASAP, we design a new approach to determine the relationship between the number of estimators N_e and error ϵ. Our approach is based on two main insights: first, we observe that for every pattern based on the probability of sampling, a loose upper bound for the number of estimators required can be computed using Chernoff bounds. For instance for triangle counting, the sampling probability is $1/mc$ where m is the number of edges and c is the degree of first chosen edge(§ 2.3.1). This probability bound can be translated to an estimator of form $N_e > \frac{K*m*\Delta}{\epsilon^2 P}$ (Theorem 3.3 [48]) where K is a constant, m is the number of edges, Δ is the maximum degree and P is the ground truth or the exact number of triangles. At a high level, the bound is based on the fact that the maximum degree vertex leads to the worst case scenario where we have the minimum probability of sampling. Similar bounds exist for 4-cliques and other patterns [48]. These theoretical bounds provide a relation between the number of estimators (N_e), error bound (ϵ) and ground truth (P) in terms of the graph properties such as m and Δ.

The second insight we use is that for smaller graphs we can get a very close approximation to the ground truth by using a very large number of estimators. This is useful in practice as this avoids having to run the exact algorithm to get a good estimate of the ground truth. Based on these two insights, the steps we follow are:

(a) We first uniformly sample the graph by edges to reduce it to a size where we can obtain a nearly 100% accurate result. In our experiments, we find that $5-10\%$ of the graph is appropriate according to the size of the graph.
(b) On the sampled graph, we run our algorithm with a large number of estimators (N_{gt}) to find $\hat{P}_s$, a value very close to the ground truth for the sampled graph.
(c) Using $\hat{P}_s$ as the ground truth value and the theoretical relationship described above, we compute the value of other variables on the sampled graph. For example, in the sampled graph, it is easy to compute m_s and Δ_s, and then infer K by running varying number of estimators.
(d) Finally we scale the values m_s, Δ_s and $\hat{P}_s$ to the larger graph to compute N_e. We note that the scaled $\hat{P}$ might not be close to P for the larger graph. But as we use the worst case bound to compute $\hat{P}_s$, the computed value of N_e offers a good bound in practice for the larger graph.

5.3 Handling Evolving Graphs

The ELP building process in ASAP is designed to be fast and scalable. Hence, it is possible to extend our pattern mining technique to evolving graphs [37] by simply rebuilding the ELP every time the graph is updated. However, in practice, we don't need to rebuild the ELP for every update. and that it is possible to reuse an ELP for a limited number of graph changes. Thus we use a simple heuristic where are a fixed number of changes, say 10% of edges, we rebuild the ELP. The general problem of accurately estimating when a profile is incorrect for approximate processing systems is hard [5] and in the future we plan to study if we can automatically determine when to rebuild the ELP by studying changes to the smaller sample graph we use in § 5.2.

6 Evaluation

We evaluate ASAP using a number of real-world graphs and compare it to Arabesque, a state-of-the-art distributed graph mining system. Overall, our evaluations show that:

- Compared to Arabesque, we find ASAP can improve performance by up to 77× with just 5% loss of accuracy for counting 3-motifs and 4-motifs.
- We find that ASAP can also scale to much larger graphs (up to 3.7B edges) whereas existing systems fail to complete execution.

Graph	Nodes	Edges	Degrees
CiteSeer [30]	3,312	4732	2.8
MiCo [30]	100,000	1,080,298	22
Youtube [41]	1,134,890	2,987,624	8
LiveJournal [41]	3,997,962	34,681,189	17
Twitter [39]	41.7 million	1.47 billion	36
Friendster [61]	65.5 million	1.80 billion	28
UK [16, 17]	105.9 million	3.73 billion	35

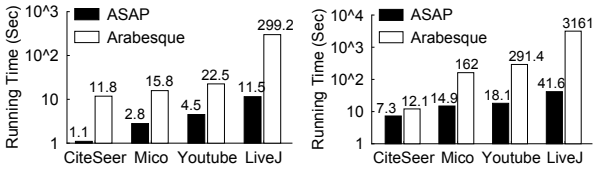
Table 2: Graph datasets used in evaluating ASAP.

- Our techniques to build error profile and time profile (ELP) are highly accurate across all the graphs while finishing within a few minutes.

Implementation. We built ASAP on Apache Spark [63], a general purpose dataflow engine. The implementation uses GraphX [34], the graph processing library of Spark to load and partition the graph. We do not use any other functionality from GraphX, and our techniques only use simple dataflow operators like `map` and `reduce`. As such, ASAP can be implemented on any dataflow engine.

Datasets and Comparisons. Table 2 lists the graphs we use in our experiments. We use 4 small and 3 large graphs and compare ASAP against Arabesque [55] (using its open-source release [2] built on Apache Giraph [14]) on four smaller graphs: CiteSeer [30], Mico [30], Youtube [41], and LiveJournal [41]. For all other evaluations, we use the large graphs. Our experiments were done on a cluster of 16 Amazon EC2 `r4.2xlarge` instances, each with 8 virtual CPUs and 61GiB of memory. While all of these graphs fit in the main memory of a single server, the intermediate state generated (§2) during pattern mining makes it challenging to execute them. Arabesque, despite being a highly optimized distributed solution, fails to scale to the larger graphs in our cluster. We note that Arabesque (or any exact mining system) needs to enumerate the edges significantly more number of times compared to ASAP which only needs to do it once or twice, depending on the query.

Patterns and Metrics. For evaluating ASAP, we use two types of patterns, *motif*s and *clique*s. For motifs, we consider 3-motifs (consisting of 2 individual patterns), and 4-motifs (consisting of 6 individual patterns) and for cliques, we consider 4-cliques. For our experiments, we run 10 trials for each point and report the median, and error bar in the ELP evaluation. We do not include the time to load the graph for any of the experiments for ASAP and Arabesque. We use total runtime as the metric when raw performance is evaluated. When evaluating ASAP on its ability to provide errors within the requested bound, we need to know the *actual* error so that it can be compared with ASAP's output. We compute actual error as $\frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth number of a specific pattern in a given graph. Since this requires us to know the ground-truth, we use simpler, known patterns, such as triangles and chains, where the ground-truth can be obtained from verified sources for such experiments. Note that the *actual error* is only used for evaluation purposes. Unless otherwise stated, the ASAP evaluations were done with an error target of 5% at 95% confidence.



(a) 3-Motif Counting **(b)** 4-Motif Counting

Figure 8: ASAP is able to gain up to 77× improvement in performance against Arabesque. The gains increase with larger graphs and more complex patterns. Y-axis is in log-scale.

6.1 Overall Performance

We first present the overall performance numbers. To do so, we perform comparisons with Arabesque and evaluate ASAP's scalability on larger graphs. We do not include ELP building time in these numbers since it is a one-time effort for each graph/task and we measure this in § 6.3.

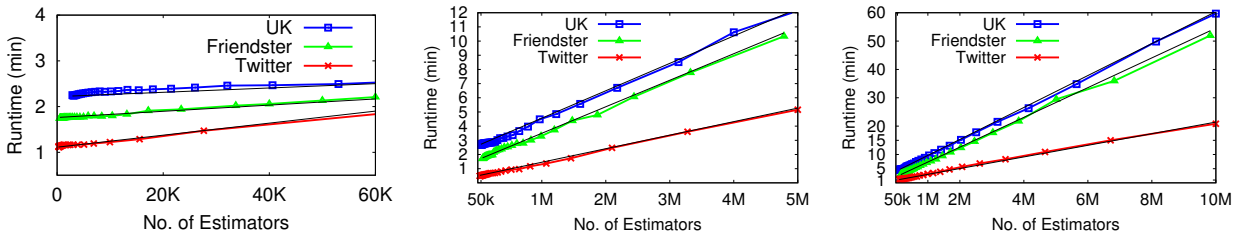
Comparison with Arabesque. In this experiment, we compare Arabesque and ASAP on the 4 smaller graphs (Table 2). In each of these systems, we load the graph first, and then warm up the JVM by running a few test patterns. Then we use each system to perform 3-motif and 4-motif mining, and measure the time taken to complete the task. In Arabesque, we do not consider the time to write the output. Similarly, for ASAP we do not output the patterns embeddings. The results are depicted in figs. 8a and 8b.

We see that ASAP significantly outperforms Arabesque on all the graphs on both the patterns, with performance improvements up to 77× with under 5% loss of accuracy. The performance improvements will increase if the user is able to afford a larger error (e.g., 10%). We also noticed that the performance gap between Arabesque and ASAP increases with larger graph and/or more complex patterns. In this experiment, mining the more complex pattern (4-motif) on the largest graph (LiveJournal) provides the highest gains for ASAP. This validates our choice of using approximation for large-scale pattern mining.

Scalability on Larger Graphs. We repeat the above experiment on the larger graphs. Since Arabesque fails to execute on these graphs on our cluster, we also provide performance numbers that were reported by its authors [55] as a rough comparison. The results are shown in Table 3.

When mining for 3-motif, ASAP performs vastly superior on the Twitter, the Friendster, and the UK graphs. Arabesque's authors report a run time of approximately 11 hours on a graph with a similar number of edges. This translates to a 258× improvement for ASAP. In the case

[4]These graph datasets in Arabesque are not publicly available.



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 9: Runtime vs. number of estimators for Twitter, Friendster, and UK graphs. The black solid lines are ASAP's fitted lines.

3-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	2.5m
	16 x 8	Friendster	66M	1.8B	5.0m
	16 x 8	UK	106M	3.7B	5.9m
Arabesque	20x32	Inst[4]	180M	0.9B	10h45m

4-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	22m
	16 x 8	UK	106M	3.7B	47m
	16 x 8	LiveJ	**4M**	**34M**	0.7m
Arabesque	16 x 8	LiveJ	**4M**	**34M**	53m
	20x32	SN[4]	**5M**	**199M**	6h18m

Table 3: Comparing the performance of ASAP and Arabesque on large graphs. The System column indicates the number of machines used and the number of cores per machine.

of 4-motifs, ASAP is easily able to scale to the more complex pattern on larger graphs. In comparison, Arabesque is only able to handle a much smaller graph with less than 200 million edges. Even then, it takes over 6 hours to mine all the 4-motif patterns. These results indicate that ASAP is able to not only outperform state-of-the-art solutions significantly, but do so in a much smaller cluster. ASAP is able to effortlessly scale to large graphs.

6.2 Advanced Pattern Mining

We next evaluate the advanced pattern mining capabilities in ASAP described in § 4.3.

Motif mining. We first evaluate the impact of ASAP's optimization when handling motif queries for multiple patterns. We use the Twitter graph and study a 4-motif query that looks for 6 different patterns. In this case ASAP caches the 3-node chain that is shared by multiple patterns. As shown in Table 4, we see a 32% performance improvement from this.

Predicate Matching. To study how well predicate matching queries work, we annotate every edge in the Twitter graph with a randomly chosen property. We then consider a 3-motif query which matches 10% of the edges. With ASAP's filtering based technique, the "all" query completes in 27 seconds, compared to 2.5 minutes when running without pre-filtering.

Accuracy Refinement. We study a scenario where the user first launches a 3-motif query on the Twitter graph with 10% error guarantee and then refines the results with another query that has a 5% error bound. We find that the running time goes from 2.5min to 1.5min (40% improvement) when our caching technique is enabled.

Pattern	Baseline	ASAP	Improv.
Motif Mining	32.2min	22min	32%
Predicate Matching	2.5min	27s	82%
Accuracy Refinement	2.5min	1.5min	40%

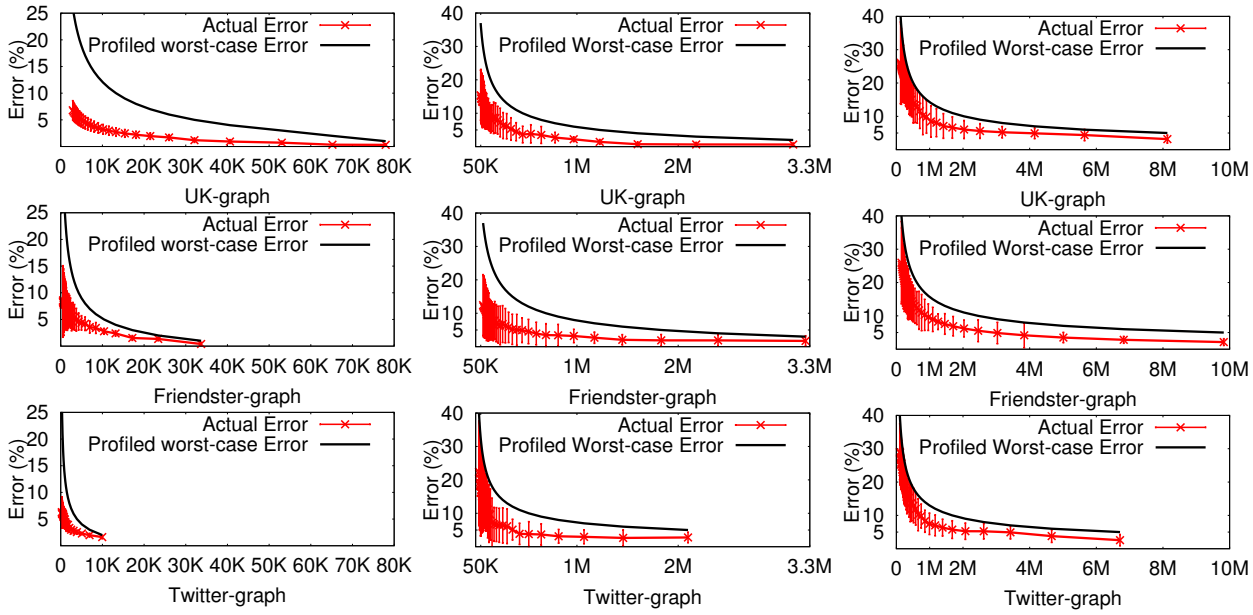
Table 4: Improvements from techniques in ASAP that handle advanced pattern mining queries.

6.3 Effectiveness of ELP Techniques

Here, we evaluate the effectiveness of the ELP building techniques in ASAP, described in §5.

Time Profile. To evaluate how well our time profiling technique (§ 5.1) works, we run three patterns—3-chains, triangles, and 4-cliques—on the three large graphs. In each graph, we obtain the time vs. estimator curve by exhaustively running the mining task with varying number of estimators and noting the time taken to complete the task. We then use our time profiling technique which uses a small number of points instead of exhaustive profiling to obtain ASAP's estimate. We plot both the curves in fig. 9 for each of the three graphs. In these figures, the colored lines represent the actual (exhaustively profiled) curve, and the black line shows ASAP's estimate. From the figure we can see that the time profile estimated by ASAP very closely tracks the actual time taken, thereby showing the effectiveness of our technique.

Error Profile. We repeat the experiment for evaluating ASAP's error profile building technique. Here, we exhaustively build the error profile by running a different number of estimators on each graph, and note the error. Then we use ASAP's technique of using a small portion of the graph to build the profile. We show both in fig. 10. We see that the actual errors are always within the estimated profile. This means that ASAP is able to guarantee that the answer it returns is within the requested error bound. We also note that in real-world graphs, the worst-case bounds are never really reached. In edge cases, where the number of patterns in the graphs are high like the chains in UK graph, the overestimation may be large, and one concern might be that we run more estimators than



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 10: Error vs. number of estimators for Twitter, Friendster, and UK graphs.

Graph	Task	Time Profile	Error Profile
UK-2007-05	3-Chain	5.2m	2.1m
	3-Motif	6.1m	2.7m
	4-Clique	9.5m	4.8m
	4-Motif	11.2m	5.9m

Table 5: ELP building time for different tasks on UK graph

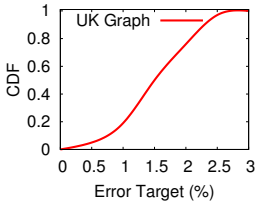


Figure 11: CDF of 100 runs with 3% error target.

required. We are working on techniques that can help us determine a tighter bound for the number of estimators in the future but as discussed in § 6.1, even with this overestimation we get significant speedups in practice. This experiment confirms that ASAP's heuristic of using a very small portion of the graph and leveraging the Chernoff bound analysis (§ 5.2) is a viable approach.

Error rate Confidence. In Figure 11, we evaluate the cumulative distribution function (CDF) of 100 independent runs on the UK graph with 3% error target and 99% confidence. We can see that 100/100 actual results are not worse than 3% error and 74/100 results are within 2% error. Thus the actual results are even better than the theoretical analysis for 99% confidence.

ELP Building Time. Finally, we evaluate the time taken for building the profiling curves. For this, we use the UK graph and configure ASAP to use 1% of the graph to build the error profile. The results are shown in table 5 for different patterns, which shows that the time to build the profiles is relatively small, even for the largest graph.

6.4 Scaling ASAP on a Cluster

ASAP partitions the graph into different subgraphs based on random vertex partition, and aggregates scaled results in the final reduce phase. In this section we evaluate how configurations with different numbers of machines impact the accuracy. In Fig. 12, we consider two scenarios: *strong-scaling*, where we fix the total number of estimators used for the entire graph, and increase the number of machines used; and *weak-scaling* where we fix the number of estimators used per-machine and thus correspondingly scale the number of estimators as we add more machines. We run the triangle counting task with the Twitter graph on different cluster sizes of 4, 8, 12, and 16 machines. From the figure we see that in the strong-scaling regime, adding more machines has no impact on the accuracy of ASAP and that we are able to correctly adjust the accuracy as more graph partitions are created. In the weak-scaling case we see that the accuracy improves as we increase more machines, which is the expected behavior when we have more estimators.

6.5 More Complex Patterns

Finally, we evaluate the generality of ASAP's techniques by applying to mine 5-motifs, consisting of 21 individual patterns. This choice was influenced by our conversations with industry partners, who use similar patterns in their production systems. Due to the complexity of the patterns, we used a larger cluster for this experiment, consisting

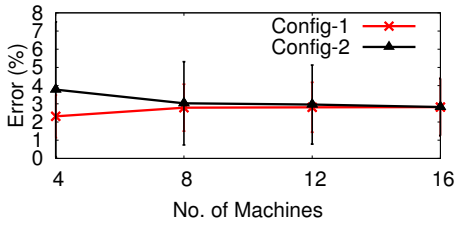


Figure 12: The errors from two cluster scenarios with different number of nodes. Config-1:*strong-scaling* to fix the total number of estimators as 2M × 128; Config-2: *weak-scaling* to fix the number of estimators per executor as 2M.

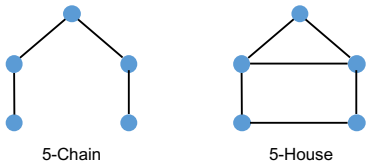


Figure 13: Two representative (from 21) patterns in 5-Motif.

of 16 machines, each with 16 cores and 128GB memory. Due to space constraints, and also because of the absence of a comparison, we only provide ASAP's performance on two representative patterns in table 6. As we see, ASAP is able to handle complex patterns on large graphs easily.

7 Related Work

A large number of systems have been proposed in the literature for **graph processing** [20, 23, 34, 35, 40, 42, 50, 53, 54, 58, 64]. Of these, some [40, 42, 54] are single machine systems, while the rest supports distributed processing. By using careful and optimized operations, these systems can process huge graphs, in the order of a trillion edges. However, these systems have focused their attention mainly on *graph analysis*, and do not support efficient graph pattern mining. Some systems implement very specific versions of simple pattern mining (e.g., triangle count). They do not support general pattern mining.

Similar to graph processing systems, a number of **graph mining** systems have also been proposed. Here too, the proposals contain a mix of centralized systems and distributed systems. These proposals can be classified into two categories. The first category focuses on mining patterns in an input consisting of multiple small graphs. This problem is significantly easier, since the system only finds one instance of the pattern in the graph, and is trivially incorporated in ASAP. Since this approach can be massively parallelized, several distributed systems exist that focus specifically on this problem. The state-of-the-art in distributed, general purpose pattern mining systems is Arabesque [55]. While it supports efficient pattern mining, the system still requires a significant amount of time to process even moderately sized graphs. A few distributed systems have focused on providing approximate pattern mining. However, these systems focus on a specific algorithm, and hence are not general-purpose.

5-Chain	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	9.2m
	16 x 16	UK	106M	3.7B	17.3m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	3.2m
	16 x 16	UK	106M	3.7B	6.5m
5-House	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	12.3m
	16 x 16	UK	106M	3.7B	22.1m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	5.6m
	16 x 16	UK	106M	3.7B	14.2m

Table 6: Approximating 5-Motif patterns in ASAP.

In distributed data processing, **approximate analysis systems** [6, 13, 32] have recently gained popularity due to the time requirements in processing large datasets. Following the approximate query processing theory in the database community, these systems focus on reducing the amount of data used in the analysis process in the hope that the analysis time is also reduced. However, as we show in this work, applying the exact algorithm on a sampled graph does not yield desired results. In addition, doing so complicates, or even makes it infeasible to provide good time or error guarantees.

Theory community has invested a significant amount of time in analyzing and proposing **approximate graph algorithms** for several graph analysis tasks [9, 10, 15, 18, 28, 33]. None of these are aimed at distributed processing, nor do they propose ways to understand the performance profile of the algorithms when deployed in the real world. We leverage this rich theoretical foundation in our work by extending these algorithms to mine general patterns in a distributed setting. We further devise a strategy to build accurate profiles to make the approach practical.

8 Conclusion

We present ASAP, a distributed, sampling-based approximate computation engine for graph pattern mining. ASAP leverages graph approximation theory and extends it to general patterns in a distributed setting. It further employs a novel ELP building technique to allow users to trade-off accuracy for result latency. Our evaluation shows that not only does ASAP outperform state-of-the-art exact solutions by more than a magnitude, but it also scales to large graphs while being low on resource demands.

Acknowledgments We thank our shepherd Roxana Geambasu and the reviewers for their valuable feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware. Liu, Braverman, and Jin are supported in part by NSF grants No. 1447639, 1650041, 1652257, 1813487, and CRII-NeTS-1755646, Cisco faculty award, ONR Award N00014-18-1-2364, and a Facebook Communications & Networking Research Award.

References

[1] Enterprise DBMS, Q1 2014. https://www.forrester.com/report/TechRadar+Enterprise+DBMS+Q1+2014/-/E-RES106801.

[2] Graph data mining with arabesque. http://arabesque.io.

[3] Graph DBMS increased their popularity by 500% within the last 2 years. http://db-engines.com/en/blog_post//43.

[4] RISELab Sponsors. https://rise.cs.berkeley.edu/sponsors/, 2018.

[5] Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., Mozafari, B., and Stoica, I. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of SIGMOD '14* (New York, NY, USA), ACM, pp. 481–492.

[6] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., and Stoica, I. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 29–42.

[7] Aggarwal, C. C., and Wang, H., Eds. *Managing and Mining Graph Data*, vol. 40 of *Advances in Database Systems*. Springer, 2010.

[8] Ahmed, N. K., Duffield, N., Neville, J., and Kompella, R. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 1446–1455.

[9] Ahn, K. J., Guha, S., and McGregor, A. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2012), SODA '12, Society for Industrial and Applied Mathematics, pp. 459–467.

[10] Ahn, K. J., Guha, S., and McGregor, A. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, NY, USA, 2012), PODS '12, ACM, pp. 5–14.

[11] Al Hasan, M., and Zaki, M. J. Output space sampling for graph patterns. *Proc. VLDB Endow. 2*, 1 (Aug. 2009), 730–741.

[12] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 469–482.

[13] Ananthanarayanan, G., Hung, M. C.-C., Ren, X., Stoica, I., Wierman, A., and Yu, M. Grass: Trimming stragglers in approximation analytics. In *NSDI* (2014), pp. 289–302.

[14] Apache Giraph. http://giraph.apache.org.

[15] Assadi, S., Khanna, S., and Li, Y. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2017), SODA '17, Society for Industrial and Applied Mathematics, pp. 1723–1742.

[16] Boldi, P., Rosa, M., Santini, M., and Vigna, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM Press, pp. 587–596.

[17] Boldi, P., and Vigna, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[18] Braverman, V., Ostrovsky, R., and Vilenchik, D. *How Hard Is Counting Triangles in the Streaming Model?* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 244–254.

[19] Bron, C., and Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM 16*, 9 (1973), 575–577.

[20] Buluç, A., and Gilbert, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25*, 4 (2011), 496–509.

[21] Buriol, L. S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohler, C. Counting triangles in data streams. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2006), PODS '06, ACM, pp. 253–262.

[22] Chaudhuri, S., Das, G., and Narasayya, V. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst. 32*, 2 (June 2007).

[23] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA), EuroSys '15, ACM, pp. 1:1–1:15.

[24] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[25] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1804–1815.

[26] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., Elmeleegy, K., and Sears, R. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 1115–1118.

[27] Cormode, G., Garofalakis, M. N., Haas, P. J., and Jermaine, C. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases 4*, 1-3 (2012), 1–294.

[28] Das Sarma, A., Gollapudi, S., and Panigrahy, R. Estimating pagerank on graph streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2008), PODS '08, ACM, pp. 69–78.

[29] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow. 7*, 7 (Mar. 2014), 517–528.

[30] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.* (2014).

[31] Fortunato, S. Community detection in graphs. *Physics reports 486*, 3 (2010), 75–174.

[32] Goiri, I., Bianchini, R., Nagarakatte, S., and Nguyen, T. D. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 383–397.

[33] Gong, N. Z., Xu, W., Huang, L., Mittal, P., Stefanov, E., Sekar, V., and Song, D. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 131–144.

[34] Gonzalez, J., Xin, R., Dave, A., Crankshaw, D., and Franklin, Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association.

[35] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[36] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[37] Iyer, A. P., Li, L. E., Das, T., and Stoica, I. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2016), GRADES '16, ACM, pp. 5:1–5:6.

[38] Jha, M., Seshadhri, C., and Pinar, A. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Trans. Knowl. Discov. Data 9*, 3 (Feb. 2015), 15:1–15:21.

[39] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[40] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 31–46.

[41] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[42] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. In *UAI* (2010), P. Grünwald and P. Spirtes, Eds., AUAI Press, pp. 340–349.

[43] Macko, P., Marathe, V. J., Margo, D. W., and Seltzer, M. I. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering* (April 2015), pp. 363–374.

[44] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. Network motifs: simple building blocks of complex networks. *Science 298*, 5594 (2002), 824–827.

[45] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[46] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[47] Pagh, R., and Tsourakakis, C. E. Colorful triangle counting and a mapreduce implementation. *CoRR abs/1103.6073* (2011).

[48] Pavan, A., Tangwongsan, K., Tirthapura, S., and Wu, K.-L. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1870–1881.

[49] Pržulj, N., Corneil, D. G., and Jurisica, I. Modeling interactome: Scale-free or geometric? *Bioinformatics 20*, 18 (Dec. 2004), 3508–3515.

[50] Quamar, A., Deshpande, A., and Lin, J. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal 25*, 2 (Apr. 2016), 125–150.

[51] Ribeiro, P., and Silva, F. G-tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov. 28*, 2 (Mar. 2014), 337–377.

[52] Robinson, I., Webber, J., and Eifrem, E. *Graph Databases*. O'Reilly Media, Inc., 2013.

[53] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 410–424.

[54] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[55] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 425–440.

[56] Tsourakakis, C. E., Kang, U., Miller, G. L., and Faloutsos, C. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 837–846.

[57] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 363–378.

[58] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[59] Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., and Zhou, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[60] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on* (2002), IEEE, pp. 721–724.

[61] Yang, J., and Leskovec, J. Defining and evaluating network communities based on ground-truth. *CoRR abs/1205.6233* (2012).

[62] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[63] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[64] Zhang, M., Wu, Y., Zhuo, Y., Qian, X., Huan, C., and Chen, K. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 608–621.

[65] Zhu, X., and Ghahramani, Z. Learning from labeled and unlabeled data with label propagation.

RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine*

Kai Wang† Zhiqiang Zuo‡ John Thorpe† Tien Quang Nguyen§ Guoqing Harry Xu†
UCLA† *State Key Laboratory for Novel Software Technology* *Facebook*§
Nanjing University‡

Abstract

Graph mining is an important category of graph algorithms that aim to discover structural patterns such as cliques and motifs in a graph. While a great deal of work has been done recently on *graph computation* such as PageRank, systems support for scalable graph mining is still limited. Existing mining systems such as Arabesque focus on distributed computing and need large amounts of compute and memory resources.

We built RStream, the *first single-machine, out-of-core* mining system that leverages disk support to store intermediate data. At its core are two innovations: (1) a rich programming model that exposes relational algebra for developers to express a wide variety of mining tasks; and (2) a runtime engine that implements relational algebra efficiently with *tuple streaming*. A comparison between RStream and four state-of-the-art distributed mining/Datalog systems — Arabesque, ScaleMine, DistGraph, and BigDatalog — demonstrates that RStream outperforms all of them, running on a 10-node cluster, *e.g.*, by at least a factor of **1.7**×, and can process large graphs on an inexpensive machine.

1 Introduction

There are two major types of analytical problems over large graphs: *graph computation* and *graph mining*. Graph computation includes a set of problems that can be represented through linear algebra over an adjacency matrix based representation of the graph. As a typical example of graph computation, PageRank [52] can be modeled as iterative sparse matrix and vector multiplications. Due to their importance in information retrieval and machine learning, graph computation problems have been extensively studied in the past decade; practical solutions have been implemented in a wide variety of graph systems [31, 27, 30, 33, 43, 39, 63, 48, 85, 58, 75, 83, 34, 57, 69, 84], most of which follow the "think like a vertex" programming paradigm pioneered by Pregel [46]. These systems have been highly optimized for locality, partitioning, and communication in order to deliver efficiency and scalability for processing very large graphs.

While this programming model makes it easy for developing computation algorithms, it is *not* designed for mining algorithms that aim to discover complex *structural patterns* of a graph rather than perform value computations. Fitting such algorithms into this model requires significant reformulation. For many mining tasks such as frequent subgraph mining (FSM), their patterns are not known *a priori*; hence, it is impossible to express these tasks using a vertex-centric model.

There is a body of work that uses declarative models to solve mining problems. Representative examples are Datalog [2, 40, 73, 62, 61], Arabesque [66], ScaleMine [4], or DistGraph [65]. For instance, due to its support for relational algebra, Datalog provides simple interfaces for developing mining tasks [40, 61]. A Datalog program for Triangle Counting, for example, needs only the following two lines of code, with *R* representing the relation of edges and *U* representing a new relation of triangles:

```
1 U(a,b,c) <- R(a,b), R(b,c), R(a,c)
2 count U(a,b,c)
```

However, Datalog's support for graph mining is rather limited since the declarative nature of its programming model dictates that only mining algorithms whose patterns are known *a priori* can be expressed by Datalog. Arabesque is a Giraph-based graph mining system that presents developers a view of "embeddings". Embeddings are subgraphs that developers can easily check to find structural patterns. Using a *filter-process* programming model, Arabesque provides full support for developing a broad set of mining algorithms. For example, Arabesque enumerates all possible subgraphs and invokes the user-defined `filter` function on each subgraph. The user logic in the function determines whether the given subgraph is an instance of the specified motif (for motif counting) or turns the subgraph into a canonical form to count the number of instances of the form (for FSM).

Specialized systems have been developed for FSM due to its broad applications. Examples are ScaleMine [4] and DistGraph [65], but these systems do not work for other mining algorithms such as Triangle Counting or Cliques.

1.1 Problems with State-of-the-Art Systems

Typical mining workloads are memory-intensive. Even simple mining algorithms can generate an enormous amount of intermediate data, which cannot fit into the main memory of any single machine. Early single-machine techniques such as gSpan [78] and GraMi [29] can analyze only small graphs as they are fundamen-

*Work was done when all authors were with UC Irvine.

tally limited by the size of the main memory of the machine on which they run. Recent mining tools such as Arabesque [66], ScaleMine [4], and DistGraph [65] are distributed systems — they leverage distributed memory resources to store intermediate mining data.

Mining Systems Distributed mining systems have several drawbacks that significantly impact their practicality. First, they commonly suffer from large startup and communication overhead. For small graphs, it is difficult for the startup/communication overhead to get amortized over the processing. For example, when FSM was executed on Arabesque to process a small graph (CiteSeer, with 4K edges) on a 10-node cluster, it took Arabesque 35 seconds to boost the system and load the graph, while executing the algorithm itself only took 3 seconds.

Second, in order to scale to large graphs, mining systems often need enterprise clusters with large amounts of memory. This is because the amount of intermediate data for a typical mining algorithm grows exponentially with the size of the graph. For example, built on top of MPI, a recent mining system DistGraph [65], using 128 IBM BlueGene/Q compute nodes, could only run 3-FSM with support = 25000[1] on a million-edge graph — even on such a small graph, the computation requires a total of 128 $\times$ 256 = 32,768GB memory. Obviously, not all users have access to such enterprise clusters. Even if they do, running a simple mining algorithm on a relatively small graph does not seem to justify very well the cost of blocking hundreds or even thousands of machines for several hours.

When many compute nodes are employed primarily to offer memory, their CPU resources are often underutilized. Unlike the "think-like-a-vertex" computation algorithms that are amenable to the bulk synchronous parallel (BSP) model, mining workloads are not massively parallel by nature — a mining algorithm enumerates subgraphs of increasing sizes to find those that match a pattern; finer-grained partitioning of the input graph to exploit parallelism often does not scale well with increased CPU resources because subgraphs often cross partitions, creating great numbers of dependencies between tasks.

Load balancing in a distributed mining system is another major challenge. Algorithms such as FSM have dynamic working sets. Their search space is often unknown in advance and it is thus hard to partition the graph and distribute the workload appropriately before the execution. When we executed FSM on DistGraph, we observed that some nodes had high memory pressure and ran out of memory in several minutes while the memory usage of some other nodes was below 10%.

[1] 25000 is a very large frequency threshold for FSM — a subgraph is considered frequent only if its frequency exceeds this threshold. The smaller the support is, the more computation is needed.

Dataflow/Datalog Systems The major problem of dataflow systems or Datalog engines is that they do not have a programming model flexible enough for expressing complex graph mining algorithms. For example, for mining frequent subgraphs whose structures have to be dynamically discovered, none of the Datalog systems can directly support it.

A Strawman Approach A possible way to develop a more cost-effective graph mining system is to add simple support for data spilling in an existing system (such as Arabesque or DistGraph) rather than developing a new system from scratch — if intermediate data can be swapped between memory and disk, the amount of compute resources needed may be significantly reduced. In fact, data spilling is already implemented in many existing systems: Arabesque is based on Giraph, which places on disk partitions that do not fit in memory; BigDatalog is based on Spark, which spills data throughout the execution. However, generic data spilling does not work well due to the lack of semantic information of how each data partition is used in the program.

To understand whether semantics-agnostic data spilling is effective, we ran transitive closure computation on BigDatalog over the MiCo graph [29] (with 1.1M edges) using a cluster of 10 nodes each with 32GB memory. Despite Spark's disk support, which spilled a total of 6.006GB of data to disk across all executors, BigDatalog still crashed in 1375 seconds.

1.2 Challenges and Contributions

To address the shortcomings of the existing mining tools, we developed RStream, the *first disk-based, out-of-core* system that supports efficient mining of large graphs. Our key insight is consistent with the recent trend on building single-machine graph computation systems [39, 58, 75, 70, 45, 83, 8, 81] — given the increasing accessibility of high-volume SSDs, a disk-based system can satisfy the large storage requirement of mining algorithms by utilizing disk space available in modern machines; yet it does not suffer from any startup and communication inefficiencies that are inherent in distributed computing.

Building RStream has two major challenges. *The first challenge* is how to provide a programming interface rich enough to support a wide variety of mining algorithms. The design of RStream's programming model is inspired from both Datalog and the gather-apply-scatter (GAS) model used widely in the existing computation systems [30, 39, 58]. On the one hand, the relational operations in Datalog enable the composition of structures of smaller sizes into a structure of a large size, making it straightforward for the developer to program mining algorithms. On the other hand, GAS is a powerful programming model that supports iterative graph processing with a well-defined termination semantics. To enable

easy programming of mining algorithms with and without statically-known structural patterns, we propose a novel programming model (§3), referred to as *GRAS*, which adds relational algebra into GAS. We show, with several examples, that under GRAS, many mining algorithms, including FSM, Triangle and Motif Counting, or Clique, can all be easily developed with less than 80 lines of code.

The second challenge is how to implement relational operators (especially join) efficiently for graphs. Since join is expensive, its efficiency is critical to the system performance. Instead of treating edges and vertices generically as relational tables as in Datalog, we take inspirations from graph computation systems to leverage the domain knowledge in graphs. In particular, we are inspired by recent systems (*e.g.*, X-Stream [58] and GridGraph [85]) that use streaming to reduce I/O costs.

The scatter/gather phase in these systems loads vertices into memory and *streams in* edges/updates to generate updates/new vertex values. The insight behind streaming is that since the number of edges/updates is much larger than the number of vertices for a graph, edge streaming provides efficiency by sequentially accessing edge data from disk (as edges are sequentially read but not stored in memory) and randomly accessing vertex data held in memory. Streaming essentially provides an *efficient, locality-aware join implementation*. RStream leverages this insight (§4) to implement relational operations.

1.3 Summary of Results

We have implemented RStream and made it publicly available at https://github.com/rstream-system. We evaluated it using 6 mining algorithms over 6 real-world graphs. With a rich programming model and an efficient implementation of the model using streaming, RStream, running on a single machine with 32GB memory and 5.2TB disk space, outperformed 4 state-of-the-art distributed mining and Datalog systems — Arabesque, ScaleMine, DistGraph, and BigDatalog by at least a factor of **1.7**×, when they each ran on a 10-node cluster.

These results do not necessarily suggest that RStream has better scalability than a distributed system, which may be able to scale to larger graphs if sufficient memory is provided. However, RStream is indeed a better choice if a user has only a limited amount of computing resources, since its disk requirement is easier to fulfill and yet it can scale to large enough real-world graphs.

2 Background and Overview

Since RStream builds on streaming, we provide a brief discussion of this idea and the related systems. We then use a concrete example to overview RStream's design.

2.1 Background

RStream's tuple streaming idea is inspired by a number of prior works, and in particular, the X-Stream graph computation system [58] that uses edge streaming to reduce I/O. X-Stream partitions a graph into *streaming partitions* based on vertex intervals. Each streaming partition consists of (1) a vertex set, which contains vertices in a logical interval and their values, (2) an edge set, containing edges whose *source vertices* are in its vertex set, as well as (3) an update set, containing updates over the edges whose *destinations* are in its vertex set. X-Stream's design is based on the GAS model. It first conducts the scatter phase, which, for each partition, loads its vertex set into memory and streams in edges from the edge set to generate updates (*i.e.*, propagate the value of the source to the destination for each edge).

The update over each edge is shuffled into the update set of the partition containing the destination of the edge. This enables an important *locality property* — for each vertex in a streaming partition, updates from all of its incoming edges are present in the update set of the same partition. The property leads to an efficient gather-apply phase, because vertex computation can be performed *locally* in each partition without accessing other partitions.

The following gather-apply phase loads the vertex set for each partition into memory, streams in updates from the update set of the partition, and invokes the user vertex function to compute a new value for each vertex. During scatter and gather-apply, edges/updates are streamed in *sequentially* from disk while in-memory vertices are randomly accessed to compute vertex values. This design leads to high performance because the number of edges is much larger than that of vertices.

2.2 RStream Overview

We use X-Stream's partitioning technique as the starting point to build RStream. RStream adds a number of relational (R) phases into the GAS programming/execution model, resulting in a new model referred to as *GRAS* in the paper. To accommodate the relational semantics, RStream's programming interface treats vertex set, edge set, and update set all as relational tables. From this point on, we use *vertex table*, *edge table*, and *update table* to refer to these sets.

Since edges do not carry data, the edge table has a fixed schema of two columns (source and destination) – its numbers of rows and columns never change. Both the vertex and update table may change their schema during computation. For example, the vertex table, initially with two columns (ID and initial value), may grow to have multiple columns (due to joins) where each vertex corresponds to a row with multiple elements; an example can be found shortly in Figure 2. In the update table, one vertex may have multiple corresponding rows since the vertex can receive values from multiple edges. The update table can also change due to joins. Tuples in these tables remain *unsorted* throughout the execution.

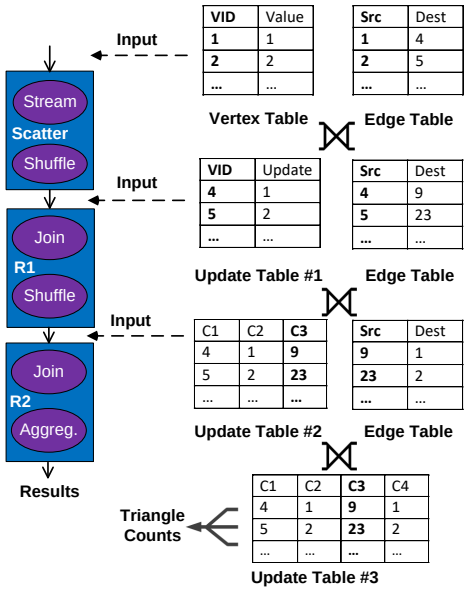


Figure 1: A Triangle Counting example in RStream; highlighted in each table is its key column. For each table, only a small number of relevant tuples are shown.

RStream first conducts scatter to generate the update table. Similarly to X-Stream, the vertex table is loaded into memory in this phase; edges are streamed in and updates are shuffled. The user-defined relational phases are then performed over the update table and the edge table in each streaming partition. What and how many relational phases are needed is programmable. These relational phases produce a new set of update tables, which will be fed as input to the gather-apply phase to compute new tuples for each vertex. The new tuples are saved into the vertex table at the end of an iteration.

Example We use Triangle Counting as an example. Although Triangle Counting is also supported by many computation systems, it is a typical structure mining algorithm that has a simple logic and thus provides a good introductory example. Figure 1 depicts the dataflow of the computation while the RStream code is shown in Figure 2. The execution contains three phases: scatter and two additional relational phases. The scatter phase has the same semantics as in X-Stream — the vertex table is loaded into memory; edges are streamed in and updates are shuffled. The relational phases are user-defined and their implementations are shown in Line 13–49. RStream lets the developer register the dataflow by connecting phases (Line 4–8). Each node on the dataflow graph is a `Phase` object. Class `TCScatter` is a scatter phase with a standard semantics; its definition is omitted for brevity. The developer adds relational phases into the dataflow.

Initially, we let the value of each vertex be its own ID (shown in the vertex table in Figure 1). The scatter

```
1  class TriangleCounting : public Application {
2    void run(Engine e){
3      /*Create a dataflow graph*/
4      TCScatter s;
5      e.set_start(&s);
6      R1 r1; R2 r2;
7      e.insert_phase(r1, s);
8      e.insert_phase(r2, r1);
9      e.run();
10   }
11 };
12
13 class R1 : public RPhase{
14   /*Called from join: only keep such <a, b, c>
         that b < a < c */
15   bool filter(Tuple t1, Tuple t2){
16     if(t1.element(1) > t1.element(0))
17        return FALSE;
18     if(t2.element(0) > t2.element(2))
19        return FALSE;
20     return TRUE;
21   }
22
23   /*Called from join: new key column*/
24   int new_key(){
25     return 2; /* set 'C3' as key*/
26   }
27
28   /*The main entry point*/
29   void execute(StreamingPartition sp){
30     UpdateTable ut = sp.update_table;
31     ut.set_key(0); //set 'VID' as key
32     EdgeTable et = sp.edge_table;
33     /*Join ut with et; et's key is 'Src';
           generated tuples are shuffled on
           new_key*/
34     super::join(sp);
35   }
36 };
37
38 class R2: public RPhase{
39   bool filter(Tuple t1, Tuple t2){
40    if(t2.element(1) != t1.element(0))
41      return FALSE;
42    return TRUE;
43  }
44
45  void execute(StreamingPartition sp){
46    super::join(sp);
47    super::aggregate(sp, COUNT, null);
48  }
49 };
```

Figure 2: Triangle counting in RStream.

phase streams edges in from the edge table. For each edge e, RStream retrieves the tuple from the vertex table corresponding to e's source vertex and produces an update based on it. In the beginning, since each vertex has only one value (*i.e.*, its own ID), the update over each edge e is essentially e's source vertex ID. These updates are shuffled into the update tables (#1 in Figure 1) across the streaming partitions. Specifically, the update for e, which is e's source vertex ID, goes into the update table of the partition that contains e's destination.

The program has two relational phases R1 and R2. R1 essentially joins all such edges (a, b) with (b, c) to produce relation (a, b, c), while R2 joins (a, b, c) with (c, a) to detect triangles. To implement R1, the developer invokes the `join` function defined in class `RPhase`. This function takes a streaming partition (sp) as input and implements a *fixed semantics* of joining sp's update table (ut) with

its own edge table (*et*) on their key columns. The key column for the update table can be set by using `set_key`, while the edge table always uses the source vertex column as its key column.

Joining the two tables also conducts (1) filtering, (2) tuple reshuffling, and (3) updating of *sp*'s update table. Filtering uses the user-defined `filter` function (Line 15–21). Tuples produced by this join form the new update table of each partition. The user can override the function `new_key` to specify the key column of this new table. If the new key is different than the current key of the update table, the generated tuples need to be reshuffled across partitions — each tuple is sent to the partition that contains the key element of the tuple.

For instance, the invocation of `join` in Line 34 joins the update table #1 with the edge table in Figure 1 using the filter defined in Line 15 of Figure 2. Specifically, it joins (a, b) with (b, c) and produces tuples of the form (a, b, c). The `filter` function specifies that we select only rows (a, b, c) with $b < a < c$, to filter out duplicates. Next, since function `new_key` specifies C3 as the new key column, each generated (a, b, c) will be shuffled to the streaming partition whose vertex table contains vertex ID c. This provides a benefit of locality for the next join, which will be performed on column C3 of the update table and Src of the edge table. Finally, the update table of each streaming partition *sp* is updated to the new table containing such (a, b, c) tuples.

The second invocation of `join` in Line 46 joins the update table resulting from R1 (*i.e.*, #2 in Figure 1) and the same edge table with the filtering condition defined in Line 39–43. The goal of this join is to find tuples of the form (a, b, c) and (c, b) to confirm that (a, b, c) indeed forms a triangle. After R2, the new update table (#3) in each partition contains triangles that can be counted using the aggregation function `aggregate` (Line 47). Here we do not need a cycle in the dataflow graph and the algorithm ends after the two joins.

Since the example aims to count the total number of triangles, a gather-apply phase is not needed. However, if one wants to count the number of distinct triangles for each vertex, an additional gather-apply phase would be required to stream in triangle tuples from the update table #3 and gather them based on their key element to compute per-vertex triangle counts. The gather phase essentially implements a group-by operation. More details can be found in §3.

Observation on Expressiveness We make several observations with the example. The first one is the expressiveness of the GRAS model. Joins performed by the relational phases over the update table and the edge table enable us to "grow" existing subgraphs we have found (*i.e.*, stored in the update table) with edges (*i.e.*, stored in the edge table) to form larger subgraphs. This is the key ability enabling Datalog and Arabesque to express mining algorithms. Our GRAS model is *as expressive as Arabesque's filter-process model* – the `filter` function in a relational phase achieves the same functionality as Arabesque's filter while Arabesque's embedding enumeration and processing can be achieved with relational joins between the update and edge tables.

Clearly GRAS is *more expressive than Datalog* – the combination of dataflow cycles and relational joins allows RStream to express algorithms that aim to discover structures whose shapes cannot be described *a priori*, such as subgraph mining.

A surprising side effect of building our programming model on top of GAS is that RStream can also support graph computation algorithms and even the transitive closure computation, which none of the existing mining systems can support. Developing computation algorithms such as PageRank is easy — they need the traditional scatter, gather, and apply, rather than any relational phases.

Observation on Efficiency The locality property of X-Stream is preserved in RStream. Tuple shuffling performed at the end of each join (based on `new_key`) makes it possible for joins to occur locally within each streaming partition *sp*. This is because (1) all the update tuples whose key column contains a vertex ID belonging to *sp* have been shuffled into the *sp*'s update table, and (2) all the edges whose source vertex (*i.e.*, key column) belonging to *sp* are already in *sp*'s edge table. Random accesses may occur only during shuffling; accesses are conducted sequentially in all other phases. Our join is implemented efficiently by tuple streaming (§4) – since the update table is often orders of magnitude larger than the edge table, RStream loads the edge table in memory and streams in tuples from the update table.

Limitation A limitation of RStream is that it currently assumes a static graph and does not deal with graph updates without restarting the computation. Hence, it cannot be used for interactive mining tasks at this moment.

3 Programming Model

This section provides a detailed description of RStream's programming model. Figure 3 shows the data structures and interface functions provided by RStream. An RStream program is made up of a dataflow graph constructed by the developer. The main entry of an RStream application is a subclass of `Application`, which the developer needs to provide to implement a given algorithm.

Adding Structural Info A special function to be implemented in an application is `need_structure`, which, by default, returns FALSE. As shown in Figure 1, each join grows an existing group of vertices with a new edge, generating a new (larger) structure. However, since each tuple currently only contains vertex IDs, the *structural*

information of these vertices (*i.e.*, edges connecting them) is missing. This will not create a problem for applications such as Triangle Counting because the structure of a triangle is known *a priori*. However, for applications like FSM, the shape of a frequent subgraph needs to be discovered dynamically. Missing structural information in tuples would create two challenges for these applications. First, tuples with identical elements may represent different structures. For example, a tuple $\langle 1,2,3,4\rangle$ may come from the joining of $\langle 1,2,3\rangle$ and $\langle 3,4\rangle$ or of $\langle 1,2,3\rangle$ and $\langle 2,4\rangle$; these are clearly two different subgraphs. The lack of structural information causes RStream to recognize them as the same subgraph instance, leading to incorrect aggregation.

Conversely, missing structural information makes it difficult for RStream to find and merge identical (automorphic) subgraphs that are represented by different tuples. For instance, joining $\langle 1,2,4\rangle$ and $\langle 2,3\rangle$ on the two columns #1 and #0 generates the same subgraph instance as joining $\langle 1,2,3\rangle$ and $\langle 2,4\rangle$ on the columns (#1, #0), although the tuples produced look different ($\langle 1,2,4,3\rangle$ and $\langle 1,2,3,4\rangle$). Failing to identify such duplicates would lead not only to mis-aggregation but also inefficiencies.

To develop applications requiring structural information, a RStream developer can override function `need_structure` to make it return TRUE. This informs RStream to append a piece of information regarding each join to each tuple produced by the join. For example, joining $\langle 1,2\rangle$ with $\langle 2,3\rangle$ on the columns (#1, #0) produces a tuple $\langle 1,2,3,(1)\rangle$, where (1) indicates that this tuple comes from expanding a previous tuple with an edge on its 2nd column.

A further join between $\langle 1,2,3,(1)\rangle$ and $\langle 2,4\rangle$ on the columns (#1, #0) generates tuple $\langle 1,2,3,4,(1,1)\rangle$, which indicates that this tuple comes from first expanding the second column with an edge and then the second column with another edge. This piece of information is added (implicitly) at the end of each tuple, encoding the history of joins, which, in turn, represents the edges that connect the vertices in the tuple.

This structural information is needed in the following two scenarios. First, it is used to encode a subgraph represented by a tuple into a coordination-free *canonical form*, which can be used by the function `is_isomorphic` (defined in `Tuple`) during aggregation to find *isomorphic subgraphs*. Two subgraphs (*i.e.*, tuples) are *isomorphic* iff there exists a one-to-one mapping between their vertices and between their edges, *s.t.* (1) each vertex/edge in one subgraph has one matching vertex/edge in another subgraph, and (2) each matching edge connects matching vertices. Tuples are aggregated at the end based on isomorphism-induced equivalence classes.

Second, the structural information is used to identify tuples representing the same subgraph instance (*i.e.*, by `is_automorphic`). Two subgraphs are *automorphic* iff they contain the same edges and vertices. Tuples that represent the same subgraph instance need to be merged during computation for correctness and performance. The implementation of these functions is discussed in §4.

```
/*Data structures*/
template <class T>
class Tuple {
  int num_elements() {...}
  T element(int i){...}
  virtual bool is_automorphic(Tuple t){...}
  virtual bool is_isomorphic(Tuple t){...}
};
class Edge : public Tuple {...};
class Vertex: public Tuple {...};

class Table {
  int get_key(){...}
  void set_key(int i) {...}
};
class UpdateTable : public Table {...};
class EdgeTable : public Table {...};
class VertexTable : public Table {...};
struct StreamingPartition {
  UpdateTable update_table;
  EdgeTable edge_table;
  VertexTable vertex_table;
  virtual void set_init_value(Vertex v);
};

class Application{
  /* Dataflow graph registered here */
  virtual void run();
  /* Whether we need structural info*/
  virtual bool need_structure() {return FALSE;}
};

/*Phases*/
class Phase {
  virtual bool converged(TerminationLogic l);
};
class Scatter : public Phase {
  virtual Tuple generate_update(Edge e){...};
};
class GatherApply : public Phase {
   virtual void apply_update(Vertex v, Tuple
        update);
};

class RPhase : public Phase{
  /* Functions called from join or select*/
  virtual bool filter(Tuple t1, Tuple t2) {
       return TRUE;}
  virtual int new_key();

 /* Called from the engine*/
  virtual void execute(StreamingPartition p);

 /* == A set of relational functions ==*/
 /* Join ut and et of p and updates ut*/
 void join(StreamingPartition p){...}
 /* Join ut and et of p on all columns of ut
      and updates ut*/
 void join_on_all_columns(StreamingPartition p)
      {...}
 /* Select rows from ut of p and updates ut*/
 void select(StreamingPartition p){...}
 /* Aggregate rows from ut of p*/
 void aggregate(StreamingPartition p, int type)
      {...}
};
```

Figure 3: Major data structures and API functions.

RStream tuples are essentially vertex-based representations of subgraphs. Edges are represented as structural

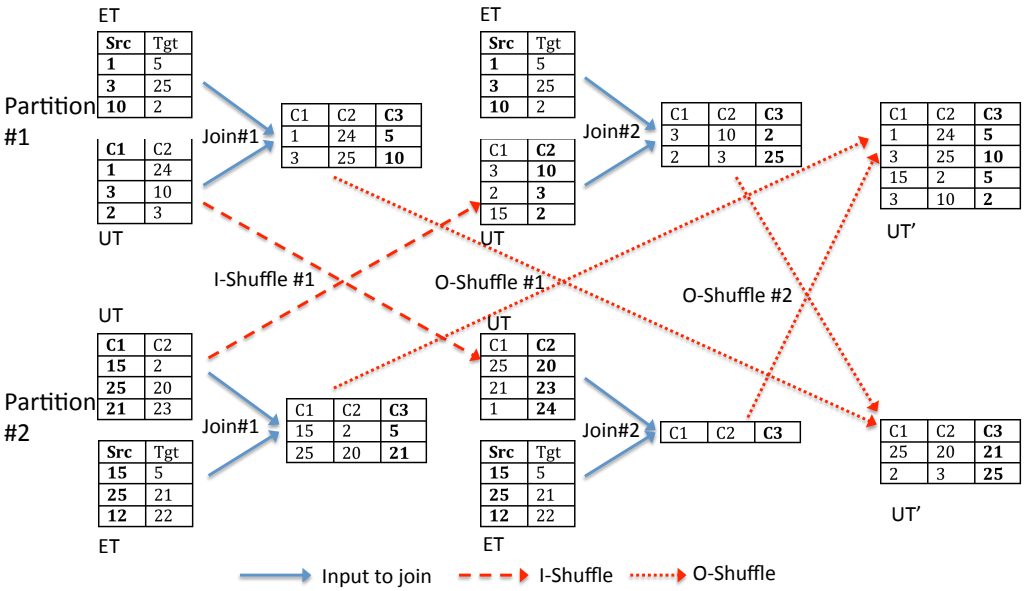


Figure 4: A graphical illustration of `join_on_all_columns`; the streaming partitions #1 and #2 contain vertices [0, 10] and [11, 25], respectively; suppose `new_key` returns 2 (which is column C_3). Structural info is not shown.

information appended at the end of each tuple. Compared to Arabesque where each subgraph (embedding) has an edge-based representation, RStream's representation allows the application to express whether the edge information is needed, providing space efficiency for applications that aim to find statically-known patterns and thus do not need the edge information.

Relational Phases Operations that can be performed in a relational phase include `join`, `select`, `aggregate`, and `join_on_all_columns`. `join` joins the update table with the edge table of each streaming partition on their key columns; `select` selects rows from the update table based on the user-defined filter; and `aggregate` aggregates values from all rows in the update table. The "type" parameter of `aggregate` indicates the type of aggregation such as MAX, MIN, SUM, COUNT, or STRUCTURE_SUM. A special type is STRUCTURE_SUM, which counts the number of subgraphs that belong to the same isomorphism class. If a programmer needs to aggregate over a subset of rows, she can first invoke `select` and then `aggregate`. `join` and `select` change the update table while `aggregate` does not. `join_on_all_columns` will be discussed shortly.

The two callback functions `filter` and `new_key` in class `RPhase` are invoked by `join`, `select`, and `join_on_all_columns` to determine what rows need to be considered and how results should be shuffled, respectively. For either `join` or `select`, changing the key column of the update table (*i.e.*, using `new_key`) will trigger tuple shuffling across streaming partitions.

Note that `RPhase` does not provide a `group-by` function, because group-by can be essentially implemented by a gather-apply phase. During a gather-apply, the vertex table is loaded into memory and tuples from the update table (produced either by a scatter phase or by a relational phase) are streamed in. RStream gathers tuples that have the same key element (*i.e.*, vertex ID) and invokes the user-defined `apply_update` function at Line 41 to compute a new tuple for the vertex. These new tuples are then saved into the vertex table, which is written back to disk at the end of each iteration. In other words, gather-apply produces a new vertex table.

`join_on_all_columns` is the same as `join` except that it joins the update table with the edge table *multiple times*, each time using a different column from the update table as key. The key of the edge table remains unchanged (*i.e.*, source vertex column). The number of joins performed by this function equals the number of columns in the update table. This function is necessary to implement mining algorithms that need to grow a subgraph from all of its vertices, such as Clique or FSM.

Figure 4 illustrates `join_on_all_columns`. Since it changes the key of the update table for each join, RStream shuffles tuples *twice* after a join — the first one, referred to as input shuffle (I-shuffle), shuffles tuples from the update table based on the next key to be used to prepare for the next join; the second one, referred to as output shuffle (O-shuffle), shuffles the result tuples based on the new key defined by `new_key` to prepare for the final output, which will eventually become the new update table (UT′).

Termination Class `Phase` contains an abstract function `converged` that needs to be implemented in user-defined phases. This function defines termination logic for iterative computation algorithms (with back edges on the dataflow graph). Note that RStream invokes this function

```
1  class FSMProgram : public Application {
2    /*FSM needs structural info*/
3    bool need_structure() { return TRUE; }
4
5    void run(Engine e){
6      Scatter cs;
7      e.set_start(cs);
8      FSMPhase fsm;
9      e.insert_phase(fsm, cs);
10     /* This forms a cycle */
11     e.insert_phase(fsm, fsm);
12     e.run();
13   }
14 };
15
16 class AggregateFilter : public RowFilter{
17   AggregationStream aggStream;
18   int threshold;
19
20   bool filter_out_row(Tuple t){
21     int support = get_support(aggStream, t);
22     if(support >= threshold) return FALSE;
23     /*It couldn't be a frequent subgraph.*/
24     return TRUE;
25   }
26 };
27
28 class FSMPhase : public RPhase{
29   static int MAX_ITE = MAX_FSM_SIZE * (
         MAX_FSM_SIZE - 1)/2;
30
31   bool converged(TerminationLogic l) {
32     if(l.get_ite_id() == MAX_ITE) return TRUE;
33     return FALSE;
34   }
35
36   int new_key(){ return LAST_COLUMN;}
37
38   void execute(StreamingPartition sp){
39     UpdateTable ut = sp.update_table;
40     ut.set_key(0);
41     EdgeTable et = sp.edge_table;
42     et.set_key(0);
43     super::join_on_all_columns(sp);
44     super::aggregate(sp, STRUCTURE_SUM);
45     AggregateFilter af;
46     super::select(sp, af);
47   }
48 };
```

Figure 5: An FSM program; structural info is needed.

only for the phases that are sources of dataflow back edges to determine whether further iterations are needed.

Example: FSM on RStream We use one more example — frequent subgraph mining — to demonstrate the power of RStream's programming model, and in particular, the usage of dataflow cycles and the function `join_on_all_columns`. Figure 5 shows the computation logic. It consists of two phases: a (standard) scatter phase and an iterative relational phase `FSMPhase`. The basic idea is that each execution of `FSMPhase` performs `join_on_all_columns` between the update and edge table. Each tuple in the update table represents a new subgraph we have found. This special join attempts to "grow" each subgraph with one edge on each vertex in the subgraph. For example, for a tuple (a,b,c,d), this join will join it with the edge table *four times*, each on a different column. Each join generates five-tuples of the form (a,b,c,d,e), which is keyed at e (*i.e.*, LAST_COLUMN specified in Line 36). Such tuples are shuffled into the partitions to which e belongs.

Given the max size of subgraphs to be considered (*e.g.*, *MAX_FSM_SIZE* = 4), all we need is to execute `FSMPhase` for a fixed number of times; this number equals the maximum number of edges that can be involved in the largest FSM: $MAX_FSM_SIZE \times (MAX_FSM_SIZE - 1)/2$, as shown in Line 29.

At the end of each `FSMPhase`, we aggregate all tuples in the update table (Line 44) to count the number of each distinct structural pattern. After the aggregation, a `select` is performed to filter out tuples corresponding to infrequent subgraphs (Line 46). This function takes as input a variable of class `AggregateFilter`, which contains a function `filter_out_row` that will be applied to each tuple. This function eliminates tuples that represent structural patterns whose supports are not high enough (Lines 20-25). The intuition here is that if a subgraph is infrequent, then any supergraphs generated based on it must be infrequent — referred to as the Downward Closure Property [7]. These infrequent tuples can be safely ignored in the next iteration. Similarly to Arabesque [66], we use the *minimum image-based support metric* [22] as it can be efficiently computed. This metric defines the frequency of a structural pattern as the *minimum* number of distinct mappings for any vertex in the pattern over all instances of the pattern.

4 RStream Implementation

RStream's implementation has an approximate of 7K lines of C++ code and is available on Github.

4.1 Preprocessing

For graphs that cannot fit into memory, they are first partitioned by a *preprocessing* step. The graph is in the edge-list or adjacency-list format on disk. RStream divides vertices into logical intervals. One interval in RStream defines a partition that contains edges whose *source vertices* fall into the interval. Edges that belong to the same partition do not need to be further sorted. To achieve work balance, we ensure that partitions have similar sizes. Since our join implementation (discussed shortly) needs to load each edge table entirely into memory, the number of streaming partitions is determined automatically to guarantee that the edge table for each streaming partition does not exceed the memory capacity while memory can still be fully utilized.

For graphs that can be fully loaded, RStream generates one single partition and no tuple shuffling will be incurred for joins. However, unlike share-memory graph computation systems that can hold all computations in memory, mining algorithms in RStream can cause update tables to keep increasing — even for very small graphs, their update tables can grow to be several orders of magnitude

larger than the size of the original graph. Hence, RStream requires disk support regardless of the initial graph size.

4.2 Join Implementation

As the update table grows quickly, to implement join, we load the edge table into memory and *stream in* tuples from the update table for each streaming partition. RStream performs *sequential disk accesses* to both the update table and the edge table, and *random memory accesses* to the loaded edge data.

Note that the edge table represents the original graph while the update table contains intermediate data generated during computation. Since the edge table never changes, the amount of memory required by RStream is bounded by the maximum size of a partition in the original graph, *not* the intermediate computation data, which can be much larger than the graph size.

Scatter and gather-apply are implemented in the same way as in X-Stream — for scatter, the vertex table is loaded while edges are streamed in; for gather-apply, the vertex table is loaded while updates are streamed in.

Filtering is performed by invoking the user-defined `filter` function upon the generation of a new tuple. When *join_on_all_columns* is used, different tuples generated may represent identical (automorphic) structures. Similarly to Arabesque, we define *tuple canonicality* by selecting a unique (canonical) tuple from its automorphic set as a representative and remove all other tuples. Details of this step are discussed shortly in §4.3.

Multi-threading RStream uses a producer-consumer paradigm for implementing join. The main thread pushes the IDs of the streaming partitions to be processed into a worklist as tasks, and starts multiple producer and consumer threads. Each producer thread pops a task off the list, loads its edge table, and streams in its update table into the producer's thread-local buffer. The producer thread joins each "old" update tuple with the edge table and produces a "new" update tuple.

We allocate a *reshuffling buffer*, for each streaming partition, to store new update tuples entering this partition. Producers and consumers synchronize using locks to ensure concurrent accesses to reshuffling buffers. Each producer sends each generated tuple to its corresponding reshuffling buffer when the buffer has room, while each consumer flushes a buffer into its corresponding "new" update table on disk when the buffer is full.

Figure 6 illustrates multiple producers and consumers. There are four producer threads and two consumer threads. Eight tasks are pushed onto the task worklist. Each producer takes one task from the list, loads its edge partition, and streams in its update partition. Each producer conducts the computation and generates output updates locally. Reshuffling is synchronized using std::mutex.

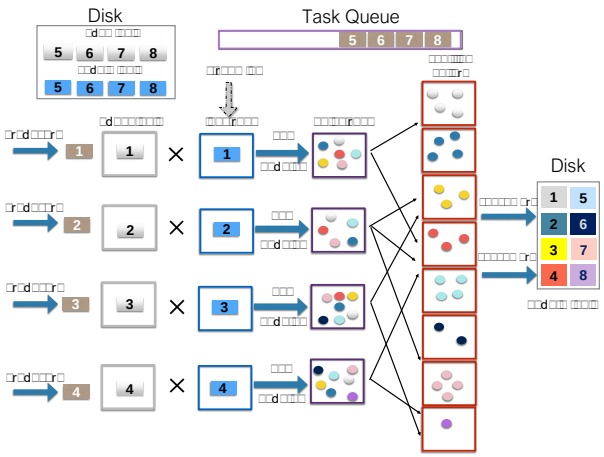


Figure 6: A graphical illustration of multiple producers, multiple consumers and reshuffling buffers.

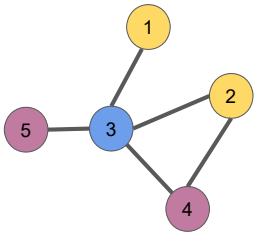
Load (Re)balancing Unlike X-Stream where the size of each streaming partition stays unchanged, in RStream, the size of each partition can grow significantly for two reasons. First, mining algorithms keep looking for graph patterns of increasing sizes, leading to the ever-growing update table. Second, tuple reshuffling at the end of each join can result in unbalanced partitions. These unbalanced partitions, if handled inappropriately, can result in significant inefficiencies (*e.g.*, underutilized CPU).

One possible solution would be to repartition the streaming partitions at the end of each relational phase for load rebalancing. However, repartitioning can incur significant disk I/O, slowing down the computation. Rather than repartition the graph, we use fine-grained tasks by dividing each update table into multiple smaller update chunks. Instead of pushing an entire update partition into the list, we push one chunk at a time. For work balancing, we also order these tasks based on their sizes so that "larger" tasks have a higher priority to be processed.

Enumeration Note that, by joining the update table with the edge table, RStream performs *breadth-first* enumeration of subgraphs. While this approach requires more storage to materialize tuples compared to a depth-first approach, it enables easier parallelization as all tuples of a given size are materialized and available for processing. Further, as a disk-based approach, RStream's breadth-first enumeration increases disk usage rather than memory usage — As shown in Figure 6, the enumeration delivers each newly generated tuple to a shuffling buffer and once the buffer is full, RStream flushes the buffer to disk.

4.3 Redundancy Removal via Automorphism Checks

Since different workers can reach identical (automorphic) tuples during processing, we need to identify and filter out such tuples. RStream adopts the idea of *embedding*



Canonical tuples of size 3		
1	3	2
1	3	4
1	3	5
2	3	4
2	3	5
3	4	5

Figure 7: A graph and its canonical tuples of size 3.

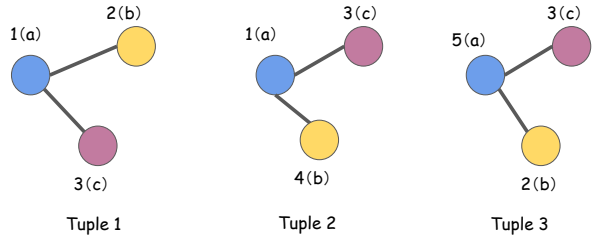


Figure 8: Aggregation example of three isomorphic tuples.

canonicality used in Arabesque [66]. We select exactly one of the automorphic tuples and elect it as "canonical". RStream runs a tuple canonicality check to verify whether a tuple t can be pruned. This algorithm runs on a single tuple without coordination. It starts with an existing canonical tuple t and checks, when t is grown with a new vertex v into a new tuple t', whether t' is also canonical. The basic idea is based on a notion of *uniqueness*: given the set S_m of all tuples automorphic to a tuple m, there exists exactly one canonical tuple t_c in S_m. The goal of this algorithm is, thus, to check whether the newly generated tuple t' is this t_c.

The tuple t' is canonical if and only if its vertices are visited in an order that is consistent with their IDs: a vertex with a smaller ID is visited earlier than one with a larger ID. In other words, RStream characterizes a tuple as the list of its vertices sorted by the order in which they are visited. When we check the canonicality of tuple t' that comes from growing an existing canonical tuple t with a vertex v, we first find the first neighbor v' of v, and then verify that there is no vertex $\in t$ after v' with a larger ID than v. Figure 7 shows a simple graph and its canonical tuples of size 3. Because RStream only processes canonical tuples, *uniqueness* is maintained in our tuple encoding (with structural information). A more detailed description can be found in [67].

4.4 Pattern Aggregation via Isomorphism Checks

For mining algorithms, aggregation needs to be done on tuples to count the number of each distinct shape (*i.e.*, structural pattern) at the end of the computation. Aggregation boils down to isomorphism checks — among all non-automorphic tuples, we count the number of those that belong to each isomorphism class. A challenge here is that isomorphism checks are expensive to compute — it is known to be isomorphism (GI)-complete and the bliss library [3] we use employs an exponential time algorithm.

RStream adopts the aggregation idea from Arabesque by turning each tuple into a *quick pattern* and then into a *canonical pattern* [16, 66]. The canonical pattern of a subgraph, which is different than the canonical tuple described earlier for automorphism checks, encodes the shape of the subgraph with all vertex information removed. Two tuples are isomorphic iff they have the same canonical patterns. The quick pattern of a subgraph is simply a total order of edges in the subgraph with vertex information removed. Two tuples may have different quick patterns even if they are isomorphic.

Given that canonical checks are expensive, we use the same two-step aggregation as in Arabesque — the first step uses quick patterns that can be efficiently computed to perform *coarse-grained* pattern aggregation, while the second step takes as input results from the first step, converts them into canonical patterns, based on which *fine-grained* aggregation is done. The aggregation conducts a two-stage MapReduce computation — the first on quick patterns and the second on canonical forms — across *all* streaming partitions. Although the aggregation idea originates from Arabesque [66], we provide a detailed example in the rest of this section to make this paper more self-contained.

Example The map phase takes quick patterns and canonical forms as input, performs local aggregation, and shuffles them into hash buckets defined by the hash value of these patterns. The reduce phase aggregates key/value pairs in the same bucket. Figure 8 depicts an example with three tuples: $tuple_1 : \langle 1(a), 2(b), 3(c), (0)\rangle$, $tuple_2 : \langle 1(a), 3(c), 4(b), (0)\rangle$, and $tuple_3 : \langle 5(a), 3(c), 2(b), (0)\rangle$. Here numbers represent vertex IDs and characters represent labels for each vertex. Note that mining algorithms often require graphs to have vertices and edges explicitly labeled. These labels represent vertex/edge properties that never change during the computation and they are needed for isomorphism checks. (0) represents the structural information obtained from the past joins.

RStream first turns each tuple into a quick pattern to reduce the number of distinct tuples. A quick pattern is obtained by simply extracting the label information and renaming vertex IDs in a given tuple, with vertex ID always starting at 1 and increasing consecutively. In the previous example, the quick patterns for the three tuples are $qp_1 : \langle 1(a), 2(b), 3(c), (0,0)\rangle$, $qp_2 : \langle 1(a), 2(c), 3(b), (0,0)\rangle$, $qp_3 : \langle 1(a), 2(c), 3(b), (0,0)\rangle$, respectively. In the map phase, RStream emits three quick pattern pairs: (qp_1, 1), (qp_2, 1), (qp_3, 1); the reduce phase further aggregates them into (qp_1, 1), (qp_2, 2) as qp_2 and qp_3 are identical.

Graphs	#Edges	#Vertices	Description
CiteSeer [29]	4,732	3,312	CS pub graph
MiCo [29]	1.1M	100K	Co-authorship graph
Patents [32]	14.0M	2.7M	US Patents graph
LiveJournal [17]	69M	4.8M	Social network
Orkut [1]	117M	3M	Social network
UK-2005 [20]	936M	39.5M	Web graph

Table 1: Real world graphs.

Program	LoC	Description
Triangle Counting (TC)	75	Counting # triangles
Closure	68	Computing transitive closure
N-Clique	36	Identify cliques of size N
N-Motif	36	Counting motifs of size N
Frequent Subgraph Mining (FSM)	40	Identify FSM of size N
Connected Components (CC)	40	Identify connected components

Table 2: Algorithms experimented.

Due to the coarse-grained modeling of quick patterns, tuples that are actually isomorphic may correspond to different quick patterns. As a next step, quick patterns are turned into canonical forms (by bliss) to perform fine-grained aggregation. A canonical form uniquely identifies a class of isomorphic subgraphs. In the example, the two quick patterns correspond to the same canonical form cf_1 : $\langle 1(a),2(b),3(c),(0,0)\rangle$. RStream eventually reports (cf_1, 3) as the final result. Since the number of quick patterns is much smaller than the number of distinct tuples, the cost of isomorphic checks can be significantly reduced.

One possible optimization is to perform *eager aggregation* — tuples are aggregated as they are being streamed into their respective partitions. We have implemented this optimization, but our experimental results showed only a minor improvement (5% in the aggregation phase and less than 2% for the overall execution).

5 Evaluation

Our evaluation focuses on three research questions:

- Q1: How does RStream compare to state-of-the-art graph mining systems? (§5.1)
- Q2: How does RStream compare to state-of-the-art Datalog engines? (§5.2)
- Q3: What is RStream's overall and I/O throughput and how quickly does data grow for mining algorithms? (§5.3)

Experimental Setup We ran our experiments using six algorithms (Table 2) over six real-world graphs (Table 1). CiteSeer, MiCo, and Patents are the graphs that were used by Arabesque and DistGraph in their evaluations. We used them primarily for comparisons with the mining systems. Similarly, Orkut and LiveJournal were used by BigDatalog [61] and we used them to compare RStream with BigDatalog. UK-2005 has almost a billion edges and is much larger than all the graphs used by Arabesque [66].

For mining algorithms, we developed Triangle Counting (TC), Clique, Motif Counting (MC), Transitive Closure Computation (Closure), and Frequent Subgraph Mining (FSM). Closure is a typical Datalog workload, and hence, we used it specifically to compare RStream with Datalog. Connected Components (CC) is a graph computation algorithm. Since RStream can also support computation (with just GAS and no relational phases), we added CC into our algorithm set to help us develop a deep understanding of the behavioral differences between graph computation and graph mining (§5.3).

		CS	MC	PA
TC	**RS**	**0.04**	**15.8**	**6.7**
	AR-10	38.1	43.1	114.9
	AR-5	39.8	44.9	116.4
	AR-1	34.2	40.7	131.5
5-C	**RS**	**0.01**	**115.1**	**35.3**
	AR-10	42.8	132.0	174.5
	AR-5	39.3	171.7	183.0
	AR-1	34.9	404.3	227.9
3-M	**RS**	**0.02**	**43.0**	**89.1**
	AR-10	40.6	51.7	116.0
	AR-5	39.7	52.8	110.5
	AR-1	32.7	47.0	132.9
4-M	**RS**	**1.41**	**93417**	**8849**
	AR-10	41.7	-	-
	AR-5	40.4	-	-
	AR-1	34.2	-	-
3-F 300	**RS**	**0.89**	**402.1**	**517.4**
	AR-10	35.9	-	-
	AR-5	39.3	-	-
	AR-1	33.7	-	-
	SM-10	2.1	69431.7	-
	SM-5	2.6	66604.3	-
	SM-1	3.5	77332.7	-
	DG-10	12.3	-	-
	DG-5	4.1	-	-
	DG-1	5.2	-	-
3-F 500	**RS**	**0.10**	**384.3**	**502.1**
	AR-10	35.7	-	-
	AR-5	39.3	-	-
	AR-1	34.4	-	-
	SM-10	2.0	15867.5	-
	SM-5	2.3	15209.4	-
	SM-1	3.2	21043.3	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.11	-	-
3-F 1K	**RS**	**0.06**	**351.7**	**383.7**
	AR-10	35.6	5790.1	-
	AR-5	39.9	5397.9	-
	AR-1	33.9	5848.2	-
	SM-10	1.2	802.6	-
	SM-5	1.1	790.8	-
	SM-1	1.1	1175.1	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.10	-	-
3-F 5K	**RS**	**0.02**	51.0	**376.4**
	AR-10	41.6	120.8	-
	AR-5	37.7	192.7	-
	AR-1	31.8	610.3	-
	SM-10	1.0	12.1	-
	SM-5	1.1	**11.6**	-
	SM-1	1.3	14.5	-
	DG-10	0.3	-	-
	DG-5	0.05	-	-
	DG-1	0.08	-	-

Table 3: Comparisons between RStream (RS), Arabesque (AR-n), ScaleMine (SM-n), and DistGraph(DG-n) on four mining algorithms — triangle counting (TC), Clique (k-C), Motif Counting (k-M), and FSM (k-F) — over three graphs CiteSeer (CS), MiCo (MC), and Patents (PA); n represents the number of nodes the distributed systems use; k is the size of the structure to be mined; '-' indicates execution failures. For FSM, four different support parameters (300, 500, 1K, and 5K) are used and explicitly shown in each 3-F row. Highlighted rows are the shortest times (in seconds).

Our experiments were conducted on a 10-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, and 3 SSDs with a total of 5.2TB disk space, running CentOS 6.8. Data was split evenly on the three disks. RStream ran on one single node with 32 threads to fully utilize CPU resources and disk bandwidth, while distributed systems used all the nodes.

5.1 Comparisons with Mining Systems

Systems and Algorithms We compared RStream with three state-of-the-art distributed mining systems: Arabesque [66], ScaleMine [4], and DistGraph [65].

Other distributed mining systems such as G-thinker [77] are not publicly available and hence not considered in our experiments. We ran these three systems with 10 nodes, 5 nodes, and 1 node to have a precise understanding of where RStream stands. In this first set of experiments, all Motif executions were run with a maximum size of 4; Clique was run with a maximum size of 5; and FSM was run with size of 3.

As discussed earlier, to run FSM we used the minimum image-based support metric [22], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all instances of the pattern. We explicitly state the support, denoted S, used in each experiment since this parameter is sensitive to the input graph. Clearly, the smaller S is, the more computation is needed.

In this experiment, we used CiteSeer, MiCo, and Patent as our input graphs. These three graphs came with labels[2] and were also used to evaluate Arabesque, ScaleMine, and DistGraph. Our initial goal was to evaluate RStream with all graphs used in prior works, but other graphs were either unavailable or do not have labels. Although these are relatively small graphs from the perspective of graph computation, running mining algorithms on them can generate orders-of-magnitude more data (see Table 5).

Table 3 reports the running times of the four systems. Note that ScaleMine and DistGraph were designed specifically to mine frequent subgraphs, and hence we could obtain only FSM's performance for these two systems. It is clear that RStream **outperforms all three systems in all cases but 3-FSM with support = 5000**. Arabesque, ScaleMine, and DistGraph failed when the size of a pattern increases. These failures were primarily due to their high memory requirement (for storing intermediate data) that could not be fulfilled by our cluster.

For FSM, on small graphs such as CiteSeer, DistGraph appears to be more efficient than the other two systems. However, DistGraph could not scale to the MiCo graph on our 10-node cluster. ScaleMine successfully processed MiCo, but took a long time, because ScaleMine trades off computation for memory; instead of caching intermediate results in memory, it always re-computes from scratch, which explains why it has better scalability but lower efficiency. None of these three systems could process FSM over the Patents graph even when support = 5000. By contrast, RStream successfully executed FSM over all the graphs under all the configurations.

RStream underperforms ScaleMine in only one case: 3-FSM (S=5000) over MiCo. RStream outperforms Arabesque (on 10 nodes) by an overall (GeoMean) of **60.9**×, ScaleMine by an overall of **12.1**×, and DistGraph by an overall of **7.2**×. As Arabesque was developed in Java, the 60.9× speedup may be partly due to RStream's use of an efficient language (C++). ScaleMine and DistGraph were both C++ applications and, hence, the wins over them provide a closer approximation of the benefit a disk-based system could offer.

[2]Mining algorithms require *labeled graphs* (*i.e.*, vertices and edges have semantic labels).

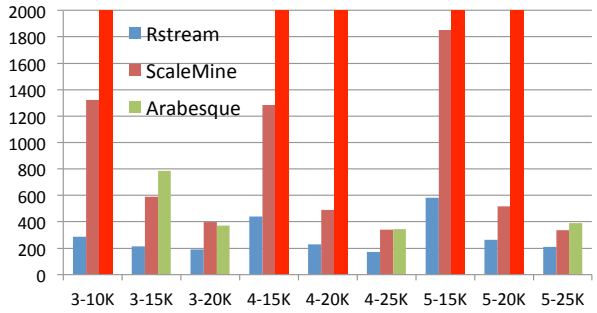


Figure 9: FSM performance comparisons with different pattern sizes and supports over the Patents graph. Tall red bars on the right of each group represent Arabesque failures.

UK Graph To understand RStream's performance on larger graphs, we ran 3-FSM on RStream to process the UK-2005 graph that has almost a billion edge. Note that none of the three distributed systems could process the graph when running 3-FSM with even a 5K support on our 10-node cluster. In all prior works, the only evidence of a mining system successfully processing a billion-edge graph was reported in [65] where DistGraph, using 512–2048 IBM BlueGene/Q machines each with 16 cores and 256GB memory, processed several synthetic graphs with 1B–4B edges in 2000 – 7000 seconds (with varying supports). Here we experimented RStream with four support parameters – 2K, 3K, 4K, and 5K – on one single machine with only 32GB memory. RStream successfully processed all of them, *e.g.*, in 4080.9, 3016.3, 2228.9, and 2146.2 seconds, respectively.

RStream ran out of memory when a relatively small support was used (*i.e.*, $\leq$1000) to compute frequent subgraphs over UK. After spending a great amount of time investigating the problem, we found that the large memory consumption was potentially due to memory leaks in the bliss library rather than RStream, which guarantees that the amount of data to be loaded from each streaming partition never exceeds the memory capacity.

Larger FSMs To evaluate how RStream performs on k-FSMs with larger k, we conducted a set of experiments over the Patents graph with various k and supports. Since DistGraph failed in most cases when we increased k, this set of experiments focused on RStream, ScaleMine, and Arabseque, and the results of the comparisons are reported in Figure 9. Both Arabesque and ScaleMine were executed with 10 nodes. Overall, RStream is **2.46**× and **2.28**× faster than ScaleMine and Arabesque.

Support	Patents		Mico	
	RStream	GraMi	RStream	GraMi
5K	**504.6**	-	**51.0**	-
10K	**286.7**	-	**23.2**	36.5
15K	**213.3**	-	**14.3**	18.7
20K	**190.8**	-	**8.6**	9.2

Table 4: FSM performance comparisons between RStream and GraMi over Patents and Mico; time is measured in seconds.

We have also compared RStream with GraMi [29], which is a specialized graph mining library designed to perform single-machine shared-memory FSM computation, over the Patents and Mico graphs. Table 4 reports the results. Note that, for each support, GraMi reports patterns of all sizes with respect to the support. RStream was executed in a similar way to provide a fair comparison. GraMi ran out of memory for all cases over the Patents graph. On the Mico graph, RStream outperforms GraMi even for large (*e.g.*, 20K) supports.

There are two reasons that could explain RStream's superior efficiency. First, joins performed by RStream grow subgraphs *in batch* while the other systems enumerate and grow embeddings individually. Second, the three systems RStream was compared against are all distributed systems that have a large startup and communication overhead. While the data size quickly grows to be larger than the memory capacity of a single machine, this size is often small in an early stage of the execution. Distributed systems suffer from communication overhead throughout the execution, while RStream does not have heavy I/O in this early stage.

The fact that the three distributed systems failed in many cases does not necessarily indicate that RStream can scale to larger graphs than them. We believe that these systems, if given enough memory, should have performed better than what is reported in Table 3. However, their exceedingly high memory requirement is very difficult to satisfy — the 10-node cluster we used is the only cluster to which we have exclusive access. According to [66], running 4-motif on a 200M-edge graph took Arabesque 6 hours consuming 20 $\times$ 110GB = 2200GB memory. As a reference point, the most memory-optimized cluster (x1.32xlarge) Amazon EC2 offers has only 1952GB memory, which is still not enough to run the algorithm.

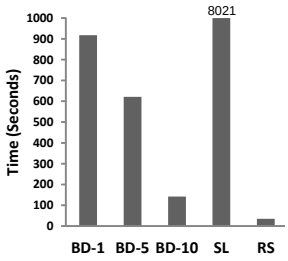
These results do suggest, though, that if a user has only a limited amount of computing resources, RStream should be a better choice than these other systems because RStream's disk requirement is much easier to fulfill and yet it can scale to large enough real-world graphs.

5.2 Comparisons with Datalog Engines

Since our GRAS model is inspired partly by the way Datalog enables easy programming of mining algorithms, we have also compared RStream with the state-of-the-art

		LiveJournal	Orkut
TC	**RS**	**87.0**	**827.4**
	BD-10	94.8	1205.3
	BD-5	109.6	1850.3
	BD-1	567.3	-
	SL	896.1	-
CC	**RS**	**101.1**	**352.0**
	BD-10	179.6	441.2
	BD-5	251.3	420.3
	BD-1	187.8	445.2
	SL	379.8	-

(a)



(b)

Figure 10: (a) Comparisons between RStream (RS), BigDatalog (BD-*n*), and SociaLite (SL) on TC and CC; (b) Closure comparison over CiteSeer.

Datalog engines. We use BigDatalog [61] with Spark joins and SociaLite [40], a shared memory Datalog engine. We used the LiveJournal and Orkut graphs, which were initially used to evaluate BigDatalog [61] to evaluate BigDatalog. We used three algorithms: Triangle Counting (TC), Connected Components (CC), and Closure Computation (Closure). Although CC and Closure are not typical mining algorithms, they are Datalog programs regularly used to evaluate the performance of a Datalog engine. Hence, we included them in this experiment. Note that BigDatalog has been shown to outperform vanilla Spark over these workloads due to several optimizations implemented over Spark joins [61].

Figure 10(a) compares the performance of RStream with that of BigDatalog and SociaLite. For TC and CC, RStream outperforms BigDatalog (with 10 nodes) by a GeoMean of **1.37**$\times$, while SociaLite failed in most cases. For transitive closure, CiteSeer was the only graph that RStream, BigDatalog, and SociaLite could all successfully process. Their performance comparison is shown in Figure 10(b): RStream is **4**$\times$ faster than BigDatalog running on 10 nodes, while it took SociaLite a large amount of time (8021 seconds) to finish closure computation.

These results appear to be different from what was reported in the prior works [61] and [40]. We found that the difference was primarily due to the input graphs — both the works [61] and [40] used synthetic acyclic graphs for transitive closure, while real graphs have both cycles and very high density that synthetic graphs do not have. Neither BigDatalog nor SociaLite could finish closure computation for any graph other than CiteSeer, while RStream successfully computed closure for LiveJournal in 4578 seconds.

5.3 RStream Performance Breakdown

To fully understand RStream's performance, throughput, I/O efficiency, and disk usage, we have conducted a set of experiments using various graphs and algorithms.

Intermediate Data Generation Table 5 reports, for 4-Motif (over the Patents graph) and 4-FSM (over the Patents graph), the number of tuples generated at the end

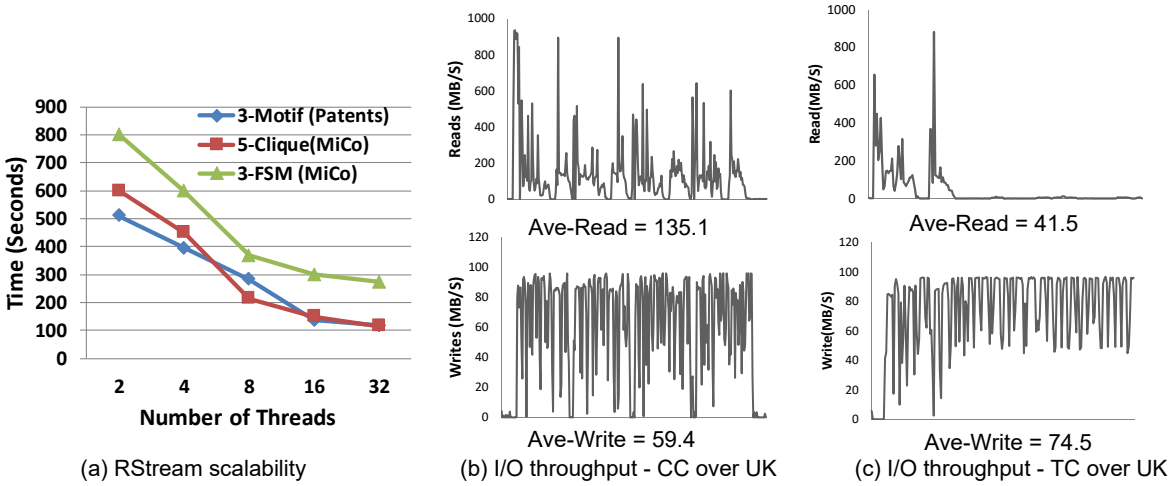


Figure 11: RStream's scalability (a), I/O throughput when running CC over UK (b), and I/O throughput when running TC over UK (c). I/O was measured with `iostat`.

	Phase	#Tuples	TS	#MB
4-Motif MiCo	0	1,080,156	16	16.5
	1	91,151,339	24	2,086.3
	2	29,044,509,725	32	886,378.1
	3	10,016,299,628	40	382,091.5
	Total	3.9×10^{10}	-	1,270,572.4 (1.21TB)
4-FSM, S=10K Patents	0	13,965,409	16	213.1
	1	625	28	0.02
	2	5,861,830	16	89.4
	3	93,313,116	24	2,135.8
	4	13,764	36	0.5
	5	29,462,761	24	674.3
	6	816,909,842	32	24,930.1
	7	101,254	44	4.2
	8	633,673,981	32	19,338.2
	9	57,361,813	40	2,188.2
	10	30,283	52	1.5
	11	509,304	40	19.4
	Total	1.65×10^{9}	-	49,594.72 (48.4GB)

Table 5: The number of tuples (**Tuples**) generated for each phase execution, the size of each tuple (**TS**), and the number of bytes (**#MB**) shuffled for 4-Motif over the Patents graph and 4-FSM, S=10K over the Mico graph.

of each phase, the size of each tuple, as well as the storage consumption of these tuples. The amount of data generated during the execution can easily exceed the memory capacity. For 4-Motif, the total amount of intermediate data generated requires 1.21TB of disk space. This motivates our out-of-core design that leverages large SSDs to store these intermediate subgraphs.

	FSM(300)	FSM(500)	FSM(1000)	3-Motif	4-Motif	5-Clique
CiteSeer	129	110	76	83	1914	26
MiCo	2388	2366	2285	1206	12408	6968
Patents	1234	1151	936	110	2791	275
UK	1367	2379	1461	1001	8914	7231

Table 6: Ratios between the final disk usage and original graph size (in the binary format).

To understand how large the total amount of data generated is, Table 6 further reports, for each graph, the ratio between the amount of storage needed at the end of each execution and the original size of the graph. This growth can be as large as 5 orders of magnitude (4-Motif over the MiCo graph). These ratios also reflect (1) the density of each graph (regardless of the size of the graph), which determines how difficult the graph is to process; and (2) the computation complexity of each algorithm, which determines how difficult the algorithm is to run. The MiCo graph is the one with the highest density, although it is relatively small in size. 4-Motif is the algorithm that needs the most computations as it generates the most intermediate data compared to other algorithms.

Scalability and I/O Figure 11(a) shows RStream's running time for varying numbers of threads. In general, RStream scales with the number of threads. However, RStream's scalability decreases when the number of threads exceeds 8 because the disk bandwidth was almost saturated when 8 threads were used.

To understand how RStream performs for mining and computation algorithms, Figure 11(b) and (c) depict RStream's I/O throughput for a computation program (CC) and a mining program (TC), respectively. For CC, we monitored I/O in a full scatter-gather-apply iteration, while for TC, our measurement covered the full cycle of a join – loading the edge table, streaming in update tuples, performing joining, and writing back to the update table. The file system cache was flushed during monitoring. Note that the high read throughput (*e.g.*, 800+MB/s)

achieved by RStream was primarily due to data stripped across the SSDs.

These two plots reveal the differences of these two types of algorithms: computation algorithms such as CC are dominated by I/O — *e.g.*, disk reads/writes occur throughout the iteration. By contrast, relational joins in the mining algorithms such as TC are more compute-intensive, as most of the reads occur in an early stage of the join and the rest of the time is all spent on the in-memory computation (of joining and aggregation). For TC, writes still scatter all over the window due to the producer-consumer model used in RStream— the number of consumer threads is often small and hence many of the disk writes overlap with the computation.

6 Related Work

RStream is the first single-machine, out-of-core graph mining system. Since graph processing is an extensively studied topic, we focus on work that is closely related.

Distributed Mining Systems Arabesque [66] is a distributed system designed to support mining algorithms. Arabesque presents to the developer an "embedding" view. Arabesque enumerates all possible embeddings with increasing sizes and the developer processes each embedding with a filter-process programming model. RStream is more efficient than Arabesque because we join tuples *in batch* rather than enumerating them individually. ScaleMine [4] is a parallel frequent subgraph mining system that contains two phases. The first phase computes an approximate solution by quickly identifying subgraphs that are frequent with high probability and collecting various statistics. The second phase computes the exact solution by using the results of the approximation to prune the search space and achieve load balancing. DistGraph [65] is an MPI-based distributed mining system that uses a set of optimizations and efficient operations to minimize communication costs. With these optimizations, DistGraph scales to billion-edge graphs with 2048 IBM BlueGene/Q nodes. G-thinker [77] is another distributed mining system that provides an intuitive graph-exploration API and a runtime engine. However, G-thinker does not support FSM and Motif-counting, which are arguably the most important mining algorithms. In addition, G-thinker's implementation is not publicly available.

Specialized Graph Mining Algorithms gSpan [78] is an efficient frequent subgraph mining algorithm designed for mining multiple input graphs. Michihiro *et al.* [38] uses an anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. GraMi [29] is an effective method for mining large input graph. Ribeiro *et al.* [55] proposes an approach for counting frequencies of motifs [54]. Maximal clique is a well-studied problem. There exist a lot of approaches to this problem, among which work from Bron-Kerbosch [23] has the highest efficiency. Recently, a body of algorithms have been developed to leverage parallel [28, 12, 59, 64], distributed systems (such as Map/Reduce) [35, 19, 41, 44, 71, 6, 36, 82, 18], or GPUs [37].

Single-Machine Graph Computation Systems Single-machine graph computation systems [39, 58, 85, 75, 42, 83, 74, 34, 70, 45, 8] have become popular as they do not need expensive computing resources and free developers from managing clusters and developing/maintaining distributed programs. State-of-the-art single-machine systems include Ligra [63], Galois [51], GraphChi [39], X-Stream [58], GridGraph [85], raphQ [75], MMap [42], FlashGraph [83], TurboGraph [34], Mosaic [45], and Graspan [74].

Ligra [63] provides a shared memory abstraction for vertex algorithms. The abstraction is suitable for graph traversal. Galois [51] is a shared-memory implementation of graph DSLs on a generalized Galois system, which has been shown to outperform their original implementations. GRACE [72], a shared-memory system, processes graphs based on message passing and supports asynchronous execution by using stale messages.

Efforts have been made to improve scalability for systems over semi-external memory and SSDs. GraphChi [39] uses shards and a parallel sliding algorithm to reduce disk I/O for out-of-core graph processing. Bishard Parallel Processor [49] reduces non-sequential I/O by using separate shards to contain incoming and outgoing edges. X-Stream [58] uses an edge-centric approach in order to minimize random disk accesses. GridGraph [85] uses partitioned vertex chunks and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. Vora *et al.* from [70] reduces disk I/O using dynamic shards.

FlashGraph [83] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [34] is an out-of-core engine for graph database to process graphs using SSDs. Pearce *et al.* [53] uses an asynchronous approach to execute graph traversal algorithms with semi-external memory. It utilizes in-memory prioritized visitor queues to execute algorithms like breadth-first search and shortest paths.

Distributed Graph Computation Systems Google's Pregel [46] provides a synchronous vertex centric framework for large scale graph processing. Many other distributed systems [46, 43, 30, 26, 57, 27, 84, 80, 60, 69, 48, 76, 24, 68] have been developed on top of the same graph-parallel abstraction.

GraphLab [43] is a framework for distributed asynchronous execution of machine learning and data mining algorithms on graphs. PowerGraph [30] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. Cyclops [26] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. Chaos [57] utilizes disk space on multiple machines to scale graph processing. PowerLira [27] is a system that dynamically applies different computation and partitioning strategies for different vertices. Gemini [84] is a distributed system that adapts Ligras hybrid push-pull computation model to a distributed form, facilitating efficient vertex-centric data update and message passing. Cube [80] uses a 3D partitioning strategy to reduce network traffic for certain machine learning and data mining problems. KickStarter [69] and Naiad [48] are systems that deal with streaming graphs.

GraphX [31] is a distributed graph system built on top of Spark, which is a general-purpose dataflow framework. GraphX provides a middle layer that offers a graph abstraction and "think like a vertex" interface for graph computation using low-level dataflow operators such as join and group-by available in Spark. GraphX's design goal is completely opposite to that of RStream— GraphX aims to *hide* the relational representation of data and operations in the underlying dataflow system to provide a user-familiar graph computation interface while RStream aims to *expose* relational representation of data and operations over the underlying graph engine to enable the expression and processing of graph mining algorithms that focus on patterns and structures.

Datalog Engines There exists a great deal of work that aims to improve efficiency and scalability for Datalog engines [13, 40, 73, 56, 61, 47]. These existing graph computation and Datalog systems are orthogonal to our work because none of them could support full graph mining. LogicBlox [13] is a system designed to reduce the complexity of software development for modern applications. It provides a LogiQL language, a unified and declarative language based on Datalog, for developers to express relations and constraints. SociaLite [40] is a Datalog engine designed for social network analysis. SociaLite programs are evaluated by parallel workers that use message passing to communicate.

Myria [73] provides runtime support for Datalog evaluation using a pipelined, parallel, distributed execution engine that evaluates a graph of operators. Datasets are sharded and stored in PostgreSQL instances at worker nodes. Both SociaLite and Myria support monotonic aggregation inside recursion using aggregate semantics based on the lattice-semantics of Ross and Sagiv [56]. BigDatalog [61] is a distributed Datalog engine built on top of Spark. It bases its monotonic aggregate (operational and declarative) semantics on work [47] that uses monotonic *w.r.t.* set-containment semantics. While RStream takes inspiration from Datalog, our experimental results show that RStream is much more efficient than existing Datalog engines on graph mining workloads.

Dataflow Systems Many dataflow systems [79, 11, 9, 21, 25] were developed. Systems such as Spark [79] and Asterix [10] provide relational operations for dataset transformations. While RStream takes inspiration from these systems, it is built specifically for graph mining, and thus has to overcome unique challenges that do not exist in existing systems.

At first glance, RStream's GRAS model is similar to a chain of MapReduce tasks — *e.g.*, the input data first gets shuffled into streaming partitions; relational expressions are next applied and the generated data is re-shuffled before the next relational phase comes. The key difference between these two model lies in the semantics — the GRAS abstraction that we built enables developers to easily develop and reason about mining algorithms by composing structures of smaller sizes into large sizes, while generic MapReduce tasks do not have any semantics. Join is a critical relational operation that has been extensively studied in the database community [5, 50, 15, 14]. While there exist many efficient join implementations such as worst-case optimal join [50], RStream is largely orthogonal to these prior works — future work could improve RStream with a more efficient join implementation.

7 Conclusion

This paper presents RStream, the first single-machine, out-of-core graph mining system. RStream employs a new GRAS programming model that uses a combination of GAS and relational algebra to support a wide variety of mining algorithms. At the low level, RStream leverages tuple streaming to efficiently implement relational operations. Our experimental results demonstrate that RStream can be more efficient than state-of-the-art distributed mining systems. We hope that these promising results will encourage future work that builds disk-based systems to scale expensive mining algorithms.

Acknowledgements

We would like to thank the many anonymous reviewers for their valuable and thorough comments. We are especially grateful to our shepherd Frans Kaashoek for his extensive feedback, helping us improve the paper substantially.

This work is supported by NSF grants CCF-1409829, CNS-1613023, CNS-1703598, and CNS-1763172, as well as by ONR grants N00014-16-1-2149, N00014-16-1-2913, and N00014-18-1-2037.

References

[1] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[2] The LogicBlox Datalog engine. http://www.logicblox.com/, 2016.

[3] Bliss: A tool for computing automorphism groups and canonical labelings of graphs. http://www.tcs.hut.fi/Software/bliss/, 2017.

[4] ABDELHAMID, E., ABDELAZIZ, I., KALNIS, P., KHAYYAT, Z., AND JAMOUR, F. ScaleMine: Scalable parallel frequent subgraph mining in a single large graph. In *SC* (2016), pp. 61:1–61:12.

[5] ABITEBOUL, S., HULL, R., AND VIANU, V., Eds. *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] ABOULNAGA, A., XIANG, J., AND GUO, C. Scalable maximum clique computation using mapreduce. In *ICDE* (2013), pp. 74–85.

[7] AGRAWAL, R., AND SRIKANT, R. Mining sequential patterns. In *ICDE*, pp. 3–14.

[8] AI, Z., ZHANG, M., WU, Y., QIAN, X., CHEN, K., AND ZHENG, W. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX ATC* (2017), pp. 125–137.

[9] ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BORKAR, V., BU, Y., CAREY, M., GROVER, R., HEILBRON, Z., KIM, Y.-S., LI, C., ONOSE, N., PIRZADEH, P., VERNICA, R., AND WEN, J. ASTERIX: An open source system for "big data" management and analysis (demo). *Proc. VLDB Endow. 5*, 12 (2012), 1898–1901.

[10] ALSUBAIEE, S., BEHM, A., BORKAR, V., HEILBRON, Z., KIM, Y.-S., CAREY, M. J., DRESELER, M., AND LI, C. Storage management in asterixdb. *Proc. VLDB Endow. 7*, 10 (2014), 841–852.

[11] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[12] APARICIO, D. O., RIBEIRO, P. M. P., AND D. SILVA, F. M. A. Parallel subgraph counting for multicore architectures. In *IPDPS* (2014), pp. 34–41.

[13] AREF, M., TEN CATE, B., GREEN, T. J., KIMELFELD, B., OLTEANU, D., PASALIC, E., VELDHUIZEN, T. L., AND WASHBURN, G. Design and implementation of the LogicBlox system. In *SIGMOD* (2015), pp. 1371–1382.

[14] ATSERIAS, A., GROHE, M., AND MARX, D. Size bounds and query plans for relational joins. In *FOCS* (2008), pp. 739–748.

[15] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. pp. 261–272.

[16] BABAI, L., AND LUKS, E. M. Canonical labeling of graphs. In *STOC* (1983), pp. 171–183.

[17] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: Membership, growth, and evolution. In *KDD* (2006), pp. 44–54.

[18] BAHMANI, B., KUMAR, R., AND VASSILVITSKII, S. Densest subgraph in streaming and MapReduce. *Proc. VLDB Endow. 5*, 5 (2012), 454–465.

[19] BHUIYAN, M. A., AND HASAN, M. A. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering 27*, 3 (2015), 608–620.

[20] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *WWW* (2004), pp. 595–601.

[21] BORKAR, V. R., CAREY, M. J., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. pp. 1151–1162.

[22] BRINGMANN, B., AND NIJSSEN, S. What is frequent in a single graph? In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD '08)* (2008), T. Washio, E. Suzuki, K. M. Ting, and A. Inokuchi, Eds., pp. 858–863.

[23] BRON, C., AND KERBOSCH, J. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM 16*, 9 (1973), 575–577.

[24] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow. 8*, 2 (Oct. 2014), 161–172.

[25] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[26] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *HPDC* (2014), pp. 215–226.

[27] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys* (2015), pp. 1:1–1:15.

[28] DI FATTA, G., AND BERTHOLD, M. R. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Trans. Parallel Distrib. Syst. 17*, 8 (2006), 773–785.

[29] ELSEIDY, M., ABDELHAMID, E., SKIADOPOULOS, S., AND KALNIS, P. GraMi: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow. 7*, 7 (2014), 517–528.

[30] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), pp. 17–30.

[31] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *OSDI* (2014), pp. 599–613.

[32] HALL, B. H., JAFFE, A. B., AND TRAJTENBERG, M. The NBER patent citation data file: Lessons, insights and methodological tools. Tech. Rep. 8498, National Bureau of Economic Research, 2001.

[33] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *EuroSys* (2014), pp. 1:1–1:14.

[34] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD* (2013), pp. 77–85.

[35] HILL, S., SRICHANDAN, B., AND SUNDERRAMAN, R. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *BCB* (2012), pp. 661–666.

[36] HUANG, Y., BASTANI, F., JIN, R., AND WANG, X. S. Large scale real-time ridesharing with service guarantee on road networks. *Proc. VLDB Endow. 7*, 14 (2014), 2017–2028.

[37] KESSL, R., TALUKDER, N., ANCHURI, P., AND ZAKI, M. J. Parallel graph mining with gpus. In *BIGMINE* (2014), pp. 1–16.

[38] KURAMOCHI, M., AND KARYPIS, G. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov. 11*, 3 (Nov. 2005), 243–271.

[39] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *OSDI* (2012), pp. 31–46.

[40] LAM, M. S., GUO, S., AND SEO, J. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE* (2013), pp. 278–289.

[41] LIN, W., XIAO, X., AND GHINITA, G. Large-scale frequent subgraph mining in MapReduce. In *ICDE* (2014), pp. 844–855.

[42] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., , AND KANG, U. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData* (2014), pp. 159–164.

[43] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[44] LU, W., CHEN, G., TUNG, A. K. H., AND ZHAO, F. Efficiently extracting frequent subgraphs using MapReduce. In *Big Data* (2013), pp. 639–647.

[45] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys* (2017), pp. 527–543.

[46] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., CZAJKOWSKI, G., AND INC, G. Pregel: A system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[47] MAZURAN, M., SERRA, E., AND ZANIOLO, C. Extending the power of datalog recursion. *The VLDB Journal 22*, 4 (Aug. 2013), 471–493.

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013), pp. 439–455.

[49] NAJEEBULLAH, K., KHAN, K. U., NAWAZ, W., AND LEE, Y.-K. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering 9*, 2 (2014), 199–212.

[50] NGO, H. Q., PORAT, E., RÉ, C., AND RUDRA, A. Worst-case optimal join algorithms: [extended abstract]. In *PODS* (2012), pp. 37–48.

[51] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *SOSP* (2013), pp. 456–471.

[52] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford University, 1998.

[53] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (2010), pp. 1–11.

[54] PRŽULJ, N. Biological network comparison using graphlet degree distribution. *Bioinformatics 23*, 2 (2007), e177–e183.

[55] RIBEIRO, P., AND SILVA, F. G-Tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov. 28*, 2 (2014), 337–377.

[56] ROSS, K. A., AND SAGIV, Y. Monotonic aggregation in deductive databases. In *PODS* (1992), pp. 114–126.

[57] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *SOSP* (2015), pp. 410–424.

[58] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP* (2013), pp. 472–488.

[59] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. In *SIGMOD* (2014), pp. 625–636.

[60] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *USENIX ATC* (2016), pp. 317–332.

[61] SHKAPSKY, A., YANG, M., INTERLANDI, M., CHIU, H., CONDIE, T., AND ZANIOLO, C. Big data analytics with datalog queries on spark. In *SIGMOD* (2016), pp. 1135–1149.

[62] SHKAPSKY, A., YANG, M., AND ZANIOLO, C. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE* (2015), pp. 867–878.

[63] SHUN, J., AND BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP* (2013), pp. 135–146.

[64] SLOTA, G. M., AND MADDURI, K. Parallel color-coding. *Parallel Comput. 47*, C (2015), 51–69.

[65] TALUKDER, N., AND ZAKI, M. J. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery: Special Issue on ECML/PKDD 2016 Journal Track Papers 30*, 5 (2016), 1024–1052.

[66] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining. In *SOSP* (2015), pp. 425–440.

[67] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining - extended version. *ArXiv e-prints* (Oct. 2015).

[68] VORA, K., GUPTA, R., AND XU, G. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim. 13*, 4 (2016), 32:1–32:27.

[69] VORA, K., GUPTA, R., AND XU, G. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS* (2017).

[70] VORA, K., XU, G., AND GUPTA, R. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC* (2016), pp. 507–522.

[71] WANG, C., AND PARTHASARATHY, S. Parallel algorithms for mining frequent structural motifs in scientific data. In *ICS* (2004), pp. 31–40.

[72] WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[73] WANG, J., BALAZINSKA, M., AND HALPERIN, D. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB 8*, 12 (2015), 1542–1553.

[74] WANG, K., HUSSAIN, A., ZUO, Z., XU, G., AND SANI, A. A. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS* (2017), pp. 389–404.

[75] WANG, K., XU, G., SU, Z., AND LIU, Y. D. GraphQ: Graph query processing with abstraction

refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC* (2015), pp. 387–401.

[76] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *SoCC* (2015), pp. 408–421.

[77] YAN, D., CHEN, H., CHENG, J., ÖZSU, M. T., ZHANG, Q., AND LUI, J. C. S. G-thinker: Big graph mining made easier and faster. *CoRR abs/1709.03110* (2017).

[78] YAN, X., AND HAN, J. gSpan: Graph-based substructure pattern mining. In *ICDM* (2002), pp. 721–.

[79] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud, p. 10.

[80] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *OSDI* (2016), pp. 285–300.

[81] ZHANG, M., WU, Y., ZHUO, Y., QIAN, X., HUAN, C., AND CHEN, K. Wonderland: A novel abstraction-based out-of-core graph processing system. In *ASPLOS* (2018), pp. 608–621.

[82] ZHAO, Z., WANG, G., BUTT, A. R., KHAN, M., KUMAR, V. S. A., AND MARATHE, M. V. SAHAD: Subgraph analysis in massive networks using hadoop. In *IPDPS* (2012), pp. 390–401.

[83] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST* (2015), pp. 45–58.

[84] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *OSDI* (2016), pp. 301–316.

[85] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015), pp. 375–386.

Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows

Vasiliki Kalavri[†], *John Liagouris*[†], *Moritz Hoffmann*[†],
Desislava Dimitrova[†], *Matthew Forshaw*[‡,*], *Timothy Roscoe*[†]

[†]*Systems Group, Department of Computer Science, ETH Zürich, firstname.lastname@inf.ethz.ch*

[‡]*Newcastle University, firstname.lastname@newcastle.ac.uk*

Abstract

Streaming computations are by nature long-running, and their workloads can change in unpredictable ways. This in turn means that maintaining performance may require dynamic scaling of allocated computational resources.

Some modern large-scale stream processors allow dynamic scaling but typically leave the difficult task of deciding *how much* to scale to the user. The process is cumbersome, slow and often inefficient. Where automatic scaling is supported, policies rely on coarse-grained metrics like observed throughput, backpressure, and CPU utilization. As a result, they tend to show incorrect provisioning, oscillations, and long convergence times.

We present DS2, an automatic scaling controller for such systems which combines a general performance model of streaming dataflows with lightweight instrumentation to estimate the *true* processing and output rates of individual dataflow operators.

We apply DS2 on Apache Flink and Timely Dataflow and demonstrate its accuracy and fast convergence. When compared to Dhalion, the state-of-the-art technique in Heron, DS2 converges to the optimal, backpressure-free configuration in a single step instead of six.

1 Introduction

We present DS2, a low-latency, robust controller for *dynamic scaling* of streaming analytics applications, which can vary the resources available to a computation so as to handle variable workloads quickly and efficiently.

Static provisioning is a poor fit for continuous, long-running streaming applications: it forces users to choose a single point on the spectrum between allocating resources for worst-case, peak load (which is inefficient) and suffering degraded performance during load spikes. Fixing resources *a priori* almost inevitably leads to a system which is over- or under-provisioned for much of its execution.

[*]Work done while visiting the Systems Group at ETH Zürich

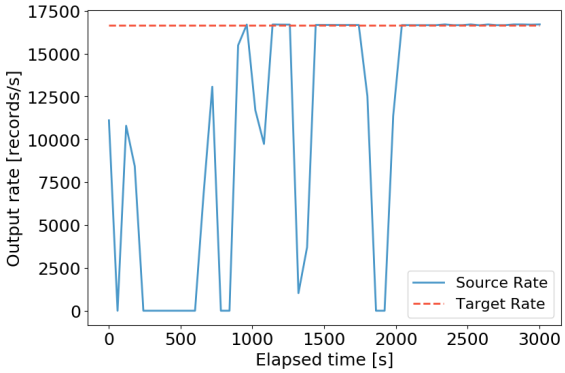


Figure 1: Effect of Dhalion's scaling decisions on the source rate when trying to match the target throughput of an under-provisioned word count dataflow.

The solution is to dynamically scale the system in response to load, an idea used extensively in cloud environments [30, 31]. This requires both a *mechanism* for scaling the computation, and a *scaling controller* which decides when and how to scale. This paper focuses on the latter; DS2 is designed to be mechanism-agnostic.

A scaling controller makes two kinds of decisions. First, it detects symptoms of over- or under-provisioning (e.g. backpressure) and decides *whether* to make a change. Detection is often straightforward and addressed by conventional monitoring tools. Second, the controller must identify the *causes* of symptoms (e.g. a bottlenecked or idle operator) and *propose* a scaling action.

The second decision is challenging, involving performance analysis and prediction. Streaming systems supporting a form of automatic dynamic scaling (e.g. Google Cloud Dataflow [26, 5], Heron [27, 13], Pravega [11], Spark Streaming [45], and IBM System S [15]) and research prototypes (e.g. Seep [12] and StreamCloud [17]) focus on the first decision and either ignore or provide speculative, often ad-hoc solutions for the second.

A good scaling controller should provide the SASO properties [19] familiar from Control Theory: *Stability* (not oscillating between different configurations), *Accuracy* (finding the optimal configuration for the given workload), *Short settling times* to reach the optimal configuration, and *not Overshooting*.

Speculative scaling decisions which do not provide these properties can be bad for streaming systems. First, they lead to temporary over- or under-provisioning, and the resulting sub-optimal resource utilization incurs unnecessary costs. Second, oscillations can in turn degrade performance due to frequent scaling actions. Finally, speculative scaling can be slow to converge, resulting in Service Level Objective (SLO) violations or load shedding.

Figure 1 illustrates these problems in the state-of-the-art Dhalion controller [13] of Heron, using the same word count dataflow as in the original paper. The dashed line shows the target throughput (source output rate), while the solid line tracks the achieved throughput, which varies due to backpressure as Dhalion changes the computation scale. Dhalion performs six scaling decisions, taking more than 30 minutes to converge.

We make the following contributions in this paper. First, we review how existing dynamic scaling techniques can lead to inaccurate, unstable, or slow provisioning decisions. We identify the causes of these effects (§ 2), which we attribute to the lack of a comprehensive performance model, dependence on heuristics, and use of coarse-grained, externally-observed execution metrics.

Second, we propose DS2, a general model and controller for automatic scaling of distributed streaming dataflows (§ 3). DS2 can accurately estimate parallelism for *all* dataflow operators within a *single* scaling decision, and operates reactively online. As a result, DS2 eliminates oscillation and overprovisioning when making scaling decisions. DS2 bases scaling decisions on real-time performance traces, and is general: it relies neither on specific signals like backpressure, as in [13], nor simplistic assumptions like 1-1 operator selectivity, as in [41].

Third, DS2 gives leverage on existing state-of-the-art approaches: when used in Heron, it identifies the optimal backpressure-free configuration in a few seconds and one step, while Dhalion performs six steps to reach an overprovisioned configuration in the same scenario (§ 5.2).

Fourth, we apply DS2 on Apache Flink (§ 5.3) and demonstrate fully-automatic scaling of streaming dataflows under dynamic workload.

Finally, we show that DS2 is accurate and converges quickly for both Apache Flink and Timely Dataflow (§ 5.4 and § 5.5). In all experiments DS2 takes at most three steps to reach the optimal configuration.

2 Background and Motivation

Designing a scaling controller with SASO properties is non-trivial, and existing dynamic scaling techniques for stream processing do not achieve them. Here, we summarize existing approaches, and then examine why they frequently lead to inaccurate, unstable, and slow scaling decisions, before proposing our solution.

Many stream processors [45, 8, 40, 27, 4, 43] have elastic runtimes and allow job reconfiguration by migrating or by externalizing state, but the majority relies entirely on manual intervention for both symptom detection and scaling actions.

Table 1 summarizes those systems that do provide some form of automatic scaling (for details also see [10]). We categorize them by (i) *metrics* used for symptom detection, (ii) *policy* logic for deciding when to scale, (iii) type of *scaling action* which defines which operators to scale and by how much, and (iv) optimization *objective* (i.e. latency or throughput SLO).

We identify two areas in which current systems fall short of the controller properties we would like: first, the metrics used do not provide enough information to make fast and accurate decisions as to how to rescale the system, and second, the policies used for scaling (and the models they are based on) are often simplistic and rule-based.

Limited metrics: Most systems rely on coarse-grained *externally observed* metrics to detect suboptimal scaling: CPU utilization, throughput, queue sizes, etc.

CPU and memory utilization can be inadequate metrics for streaming applications, particularly in cloud environments due to multi-tenancy and performance interference [38]. StreamCloud [17] and Seep [12] try to mitigate the problem by separating user time and system time, but preemption can make these metrics misleading: high CPU usage by a task running on the same physical machine as a dataflow operator can trigger incorrect scale-ups (false positives) or prevent correct scale-downs (false negatives), for example. Google Cloud Dataflow [26] uses CPU utilization only for scale-down decisions but could still suffer from false negatives. CPU usage is also unsuitable for systems like Timely [32, 33], where operators spin waiting for input.

These metrics also imply continuous threshold tuning, a cumbersome and error-prone process. Incorrect scaling decisions can often arise from slightly misconfigured thresholds, even on fine-grained metrics [13].

Dhalion [13] and IBM Streams [15] also use backpressure and congestion to identify bottlenecks. These signals are only helpful where a bottleneck exists. If the dataflow is using resources unnecessarily, such metrics will not trigger reconfiguration. Moreover, in under-provisioned dataflows, backpressure will only detect a single bottle-

System	Metrics	Policy	Scaling Action	Objective
Borealis [3]	CPU, network slack, queue sizes	Rule-based	Load shedding	Latency, throughput
StreamCloud [17]	Average CPU, observed rates	Threshold-based	Speculative, multi-operator	Throughput
Seep [12]	User/system CPU time	Threshold-based	Speculative, single-operator	Latency, throughput
IBM Streams [15]	Congestion, observed rates	Threshold-based, blacklisting	Speculative, single-operator	Throughput
FUGU+ [18]	CPU, processing time	Threshold-based	Speculative, single-operator	Latency
Nephele [29]	Mean task latency, service time, interarrival time, channel latency	Queuing theory model	Predictive, multi-operator	Latency
DRS [14]	Service time, interarrival time	Queuing theory model	Predictive, multi-operator	Latency
Stela [44]	Observed rates	Threshold-based	Speculative, single-operator	Throughput
Spark Streaming [1, 2]	Pending tasks	Threshold-based	Speculative, multi-operator	Throughput
Google Dataflow [6]	CPU, backlog, observed rates	Heuristics	Speculative, multi-operator	Latency, throughput
Dhalion [13]	Backpressure, queue sizes, observed rates	Rule-based, blacklisting	Speculative, single-operator	Throughput
Pravega [11]	Observed rates	Rule-based	Speculative, single-operator	Throughput
DS2	True processing and output rates	Dataflow model	Predictive, multi-operator	Throughput

Table 1: Overview of automatic scaling policies in distributed dataflow systems.

neck; for this reason and to minimize the effects of incorrect decisions [39, 13], each scaling action only configures one operator, increasing convergence time.

Simplistic performance models: scaling policy is generally expressed in simple rules, using predefined thresholds and conditions, e.g. *CPU utilization* $>$ *50 and backpressure* $\implies$ *scale up*. This results in a simple performance model with poor predictive accuracy, which is unable to consider the structure of the dataflow graph or computational dependencies among operators. We note the exceptions of Nephele [29] and DRS [14], which use queuing theory models. Both systems show poor prediction quality in some cases, while Nephele also seems to suffer from temporary over-provisioning and slow convergence.

Since the controller cannot accurately estimate how much to scale an operator, scaling actions are mostly *speculative*. The system applies pessimistic strategies which introduce only small changes to the number of provisioned resources [12, 15] and most policies configure a single operator at a time. This delays convergence to a steady state significantly, as all steps of the scaling process are repeated many times: SLO monitoring, decision making, state migration, and redeployment. [13] shows that, from the point that backpressure is observed, Heron needs almost an hour to reach a steady state that can handle the input rate.

More aggressive strategies apply configurations, blacklisting them if they degrade performance. [39] allows arbitrary scaling steps but requires a user-defined function to calculate the new number of instances whereas [2] supports exponential increase in resources. StreamCloud [17] tries to estimate the optimal number of VMs in a single step, but using very coarse-grained scaling (a subgraph of the dataflow topology). Google Cloud Dataflow is the only system we know with fully automatic scaling per operator, although the details of the model used have not been disclosed.

A better approach: stepping back, it seems a more promising approach for making scaling decisions would take into account both (i) each operator's *true* processing and output capabilities, regardless of backpressure or other effects, and (ii) the dataflow topology and how scaling each operator will affect downstream operators.

Figure 2 gives an intuition of how this works showing the execution timelines of operator instances in a simple dataflow. Solid lines show *useful* work performed by an instance (e.g. record processing) while dotted lines show it waiting for input or output. Edges across timelines represent data transfer.

In this example, o_1 is a bottleneck slowing down both the source and o_2 by pausing their execution. Backpressure means that an external observer sees o_1 processing 10 rec/s and o_2 processing 100 rec/s. Based on this, a policy might provision three additional instances for o_1 to reach a target of 40 rec/s, but it could not accurately estimate how much to scale o_2 and would need to make a speculative decision or apply an extra reconfiguration step.

A better approach would measure the useful time of an operator's timeline and would determine the true rate of o_1 as 10 rec/s and that of o_2 as 200 rec/s, inferring that when increasing the parallelism of o_1 to 4, it also needs to double the parallelism of o_2 to keep up with the output rate. Note this can be calculated globally, i.e. for all operators in the dataflow, *in a single step*.

DS2 does precisely this, obtaining rate measurements of each operator by lightweight instrumentation (already present in many streaming systems). In the rest of the paper we define this notion, extend it to more complex

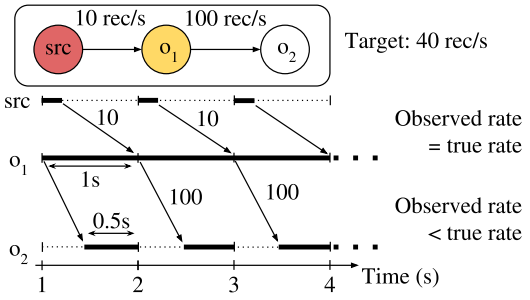


Figure 2: An under-provisioned dataflow and the execution timelines of its operators. Target throughput is 40 rec/s, but o_1 is a bottleneck creating backpressure and limiting the observed source rate to 10 rec/s.

dataflow graphs with multiple sources, and show how DS2 implements it to provide fast, accurate, and stable reconfiguration of streaming dataflows.

3 The DS2 model

DS2 identifies the optimal level of parallelism for each operator in the dataflow on the fly, while the computation executes, based on real-time performance traces. It maintains a changing *provisioning plan*, i.e. the number of resources allocated to each operator. It therefore works *online* and in a *reactive* setting.

Note that we do not target offline computation of an initial resource provisioning plan (as in [7]). Such initial configurations quickly become sub-optimal in a live system where workloads and/or internal operator states change continuously. However, for static workloads known *a priori*, DS2 could use historical performance metrics and offline micro-benchmarks (as in [20, 21, 16]) to estimate the optimal levels of parallelism before deployment.

In this section we define the scaling problem (§ 3.1), describe the DS2 model (§ 3.2), and discuss the model assumptions (§ 3.3) and properties (§ 3.4).

3.1 Problem definition

We target distributed streaming dataflow systems like Flink [9] and Heron [27] that execute data-parallel computations on shared-nothing clusters. Such a computation can be represented as a *logical* directed acyclic graph $G = (V, E)$, where vertices in V denote operators and edges in E are data dependencies between them. A vertex with no incoming edges (no *upstream* operators) is a *source* and a vertex with no outgoing edges (no *downstream* operators) is a *sink*.

A dataflow computation runs as a *physical* execution plan which maps dataflow operators to provisioned compute resources (or workers). Let the graph $G' = (V', E')$ represent the execution plan. Vertices in V' are *operator* (or *task*) instances of a corresponding vertex in V and edges are data *channels*. The assignment of tasks to workers is system-specific. We show in § 5 that DS2's scaling policy is independent of this assignment.

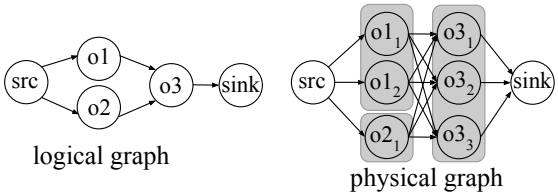


Figure 3: Logical and physical dataflow graphs.

Figure 3 illustrates a logical graph and its corresponding physical graph for a dataflow with a source, a sink, and three operators. Operators $o1$, $o2$ and $o3$ execute with two, one and three instances, respectively.

The Scaling Problem. Given a logical dataflow with sources $s_1, s_2, ..., s_n$ and rates $\lambda_1, \lambda_1, ..., \lambda_n$, identify the minimum parallelism π_i per operator such that the physical dataflow can sustain all source rates.

Source operators generate records at a *rate* λ_s, defined by application data sources (sensors, stock market feeds, etc.). To maximize system throughput, the execution plan must sustain the full source rate. This means that each operator must be able to process data without stalling its upstream operators from producing output.

Like any controller, DS2 targets workload changes on a timescale greater than its convergence time, and reacting to spikes or other changes on a shorter timescale than the convergence time would cause inefficient fluctuations. In these latter cases, the use of backpressure, buffering, or load shedding leads to more stable results than dynamic scaling at the cost of increased latency or lost data.

3.2 Performance model

We consider operator instances as repeatedly performing three activities in sequence: *deserialization*, *processing*, and *serialization*. This fits all types of operators in most modern streaming dataflow systems, including Heron, Flink, and Timely. When an operator instance is scheduled for execution, it pulls records from its input, deserializes them, applies its processing logic, and serializes the results (if any), which are pushed to the output. Serialization and deserialization are optional and happen only when data is moved between operator instances executed within different OS processes, otherwise data is usually exchanged via shared memory (e.g. queues).

The model is based on the concept of *useful time*, which we define for an operator instance as follows:

Useful Time. The time spent by an operator instance in deserialization, processing, and serialization activities.

Useful time excludes time spent waiting on input or output. Such waiting does occur in practice, for different reasons depending on the design of the reference system. In Flink, an operator instance may block on input when the input buffers are empty, or on output when there is no free space in the (bounded) output buffers. In Timely, operator instances may continuously "spin" checking their input queues until new records appear. In Heron, instances may be forced to wait due to a backpressure signal from a slow downstream operator.

In all cases, the useful time amounts to the time an operator instance runs for if executed in an *ideal* setting where it never has to wait to obtain input or push output. In general, useful time differs from the total observed time the instance needs to process and output records, and plays a key role in solving the problem of § 3.1.

Based on this distinction, we define the *true* processing and output rate of an operator instance as follows:

True Rates. The true processing (resp. output) rate corresponds to how many records an operator instance can process (resp. output) per unit of useful time.

Intuitively, the true rates denote the capacity of the operator instance, i.e. the *maximum* processing and output rate the instance could sustain for the current workload. In contrast, the observed rates are those measured by simply counting the number of records processed and output by the instance over a unit of elapsed time, which might include waiting. More precisely:

Observed Rates. The observed processing (resp. output) rate corresponds to how many records an operator instance processes (resp. outputs) per unit of observed time.

Although the observed rates are more sensitive to changing workloads, due to the potential change in waiting time, true rates typically have lower variance, especially within short time periods (e.g. a few seconds of execution) as they represent the average "cost" to process and output a single record. This cost naturally can depend on factors like the size of the record, its content, and the state maintained by the operator instance, but the average cost can be estimated using appropriate instrumentation of the operator without needing to saturate it.

We define all rates in our model relative to windows of size W seconds of observed time. We denote the useful time for an operator instance W_u, where $0 \leq W_u \leq W$. More precisely:

$$\lambda_p = \frac{R_{\text{prc}}}{W_u} \quad (1) \qquad \lambda_o = \frac{R_{\text{psd}}}{W_u} \quad (2)$$

$$\widehat{\lambda}_p = \frac{R_{\text{prc}}}{W} \quad (3) \qquad \widehat{\lambda}_o = \frac{R_{\text{psd}}}{W} \quad (4)$$

Symbol	Description
G	logical dataflow graph
m	number of operators in G ($m > 1$)
n	number of source operators in G ($0 < n < m$)
W	size of a window in time units (observed time)
W_u	useful time for an operator instance in W
R_{prc}	number of records pulled from the input in W
R_{psd}	number of records pushed to the output in W
$\widehat{\lambda}_p$	observed processing rate of an operator instance
$\widehat{\lambda}_o$	observed output rate of an operator instance
λ_p	true processing rate of an operator instance
λ_o	true output rate of an operator instance
o_i	i-th operator in G (in topological order)
p_i	number of instances of the i-th operator
$o_i[\lambda_p]$	aggregated true processing rate of the i-th operator
$o_i[\lambda_o]$	aggregated true output rate of the i-th operator
π_i	optimal number of instances for the i-th operator

Table 2: Notation used in this paper.

where λ_p and λ_o are the true processing and output rate respectively (undefined when $W_u = 0$), $\widehat{\lambda}_p$ and $\widehat{\lambda}_o$ are the observed processing and output rates (undefined when $W = 0$), and R_{prc} (resp. R_{psd}) is the total number of records the instance processed (resp. pushed) in W.

For a specific operator instance and a window W, the following inequalities hold: $0 \leq \widehat{\lambda}_p \leq \lambda_p$ and $0 \leq \widehat{\lambda}_o \leq \lambda_o$, since $0 \leq W_u \leq W$. In general, the less an operator instance waits on its input and output the smaller the difference between the observed and true rates. Table 2 summarizes the notation.

We instantiate the model with (i) the logical dataflow graph G, (ii) the output rate of each data source, and (iii) the true processing and output rates (λ_p and λ_o) of each operator instance. G is static (known at compile time) and does not change during execution, since the logical dataflow is unaffected by the scaling decisions. The output rates of the data sources are continuously monitored outside the reference system, and the true rates of the operator instances are computed based on system-generated traces, as we explain in § 4.1. The output of DS2 is the optimal parallelism, i.e. number of instances, for each logical operator in the graph G, subject to the constraints of the problem in § 3.1.

The calculation proceeds as follows: let A be the adjacency matrix of G. $A_{ij} = 1$ iff the i-th operator outputs to the j-th operator, otherwise $A_{ij} = 0$. We consider operators numbered in topological order from $i = 0$ to $i = m-1$, where m is the total number of operators in G. This means that if o_i outputs to o_j and, hence, $A_{ij} = 1$, then $0 \leq i < j < m$. Since G is acyclic (cf. § 3.1), there is a topological ordering of its nodes and it can be computed in linear time.

For a time window W and operator o_i with p_i

instances, $p_i \geq 1$, we define the *aggregated* true processing and output rates $o_i[\lambda_p]$ and $o_i[\lambda_o]$ as:

$$o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \quad (5) \qquad o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k \quad (6)$$

where λ_p^k and λ_o^k are the true processing and output rates of the k-th instance of o_i, as given by Eq. 1 and Eq. 2.

The optimal level of parallelism π_i for an operator o_i is now computed using the ratio of the aggregated true output rate of its upstream operators (when they keep up with their inputs) to the average true processing rate per instance of o_i. More formally:

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m \quad (7)$$

where m is the total number of operators in G, and n is the number of source operators in G, $0 < n < m$.

$o_j[\lambda_o]^*$ denotes the aggregated true output rate of an operator o_j, when o_j itself and all operators before it (in topological order) are deployed with their optimal parallelism to keep up with their inputs. It is recursively computed as follows:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \leq j < n \\ \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{\forall u: u<j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases} \quad (8)$$

where λ_{src}^j is the output rate of the j-th source operator, $0 \leq j < n$.

Note that $o_j[\lambda_o]^*$ depends on (i) the ratio $\frac{o_j[\lambda_o]}{o_j[\lambda_p]}$, which denotes the selectivity of o_j, and (ii) the estimated true output rate of the upstream operators ($\forall u : u < j$ in the summation). The latter implies that $o_j[\lambda_o]^*$ and, hence, π_i can be efficiently computed for *all* operators in the dataflow with a *single traversal* of G, starting from the sources. This property is important in practice, as it allows us to estimate the required number of instances for all operators in the dataflow in the same scaling decision.

3.3 Assumptions

DS2 makes the following assumptions about the dataflow system it is controlling:

Data-parallel operators. An operator's output can be produced by partitioning its input on a key and applying the operator logic separately to each partition. Other than this, the operator's internal logic can be any user-defined function. Data-parallelism is essential for effective scaling decisions: executing multiple operator instances entails partitioning its state into chunks of data processed in parallel. In contrast, non-data-parallel operators do not benefit from scaling. System users could tag such operators for DS2 to ignore, or their lack of parallelism could be identified online by comparing input and output rates before and after scaling. As with existing systems, we leave the integration of such operators for future work.

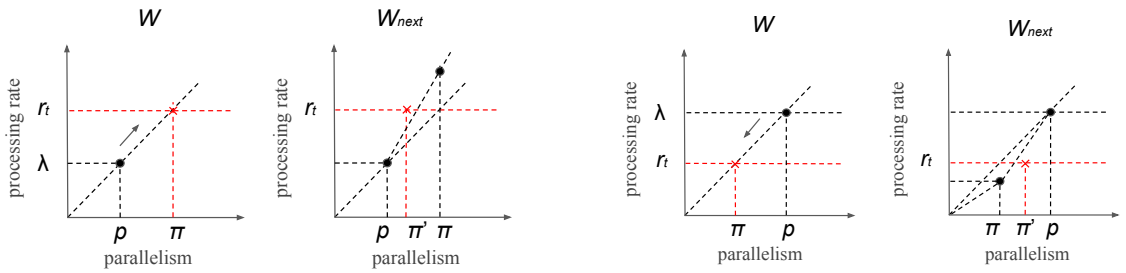
No data or computation imbalance. Our scaling model addresses neither data skew across operator instances nor computational stragglers. Both these types of imbalance can trigger backpressure which cannot be tackled by changing the degree of parallelism of one or more operators. Several robust solutions to the skew and straggler problems exist and have been incorporated into real systems. Techniques such as partial key grouping [35] introduced in Storm [34] and further evaluated in [25], and work-stealing for straggler mitigation in MapReduce [28] and Google Dataflow [26] are complementary to DS2. In § 4.2 we describe how DS2 could be integrated in a general controller for streaming applications which would not only handle dynamic scaling but also include skew and straggler handling components.

Stable workloads during scaling. Like existing scaling mechanisms, DS2 operates with the understanding that workload characteristics remain stable between a scaling decision being made and the new parallelism configuration being deployed. This window is the time taken for DS2 to make a decision (which we evaluate in § 5) plus the time to deploy the new configuration, which depends on the dataflow system in use. In practice, we find this timescale is dominated by the latter in current systems.

3.4 Properties

DS2 estimates the optimal parallelism for each operator assuming perfect scaling, that is, the true processing and output rates change linearly with the number of instances. In general, however, true rates are described by non-linear, most commonly sub-linear functions. Super-linear speedups are possible [16] (e.g. when state fits in cache after a scale-up) but are rare in practice. When this "perfect scaling" assumption holds, DS2 estimations (Eq. 7) correspond to bounds and the model enjoys the following two properties:

Property 1. No overshoot: a scale-up decision will not result in over-provisioning. The estimated optimal number of instances π_i for an under-provisioned operator is always less than or equal to the minimum required to keep up with the target rate $r_t = \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Eq. 7.



(a) No overshoot when scaling up

(b) No undershoot when scaling down

Figure 4: Given a target rate r_t and aggregated true processing rate λ which does not scale super-linearly, our model guarantees no over-provisining when scaling up and no under-provisioning when scaling down.

Property 2. No undershoot: a scale-down decision will not result in under-provisioning (and, hence, backpressure). The estimated optimal number of instances π_i for an over-provisioned operator is always greater than or equal to the minimum needed to keep up with the target rate $r_t = \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Eq. 7.

Figure 4 shows hypothetical scale up and scale down scenarios, each during two consecutive time windows, W and W_{next}. Consider an operator initially configured with parallelism p and aggregated processing rate $\lambda < r_t$, where r_t is the target rate, as shown in Figure 4a (left). Assuming linear scaling, our model assigns π instances to reach the target rate r_t. Property 1 states that there exists no $\pi' < \pi$ such that π' matches r_t. Indeed such a π' can only exist in W_{next} if the aggregated processing rate scales super-linearly, as shown in Figure 4a (right).

Similarly, if an operator is initially configured with parallelism p and aggregated processing rate $\lambda > r_t$, as in Figure 4b (left), our model assigns $\pi < p$ instances to scale down to r_t. Property 2 states that there exists no $\pi' > \pi$ such that π' matches the target r_t. As shown in Figure 4b (right), such a π' would violate the assumption of non-superlinear aggregated true processing rate.

Together, these properties imply that repetitive applications of DS2 do not oscillate: they will monotonically converge to the target rate from below or above, ensuring stability without the need to blacklist previous decisions, and simplifying the scaling mechanism significantly.

When true rates are linear and the target rate r_t is accurately estimated for each operator, DS2 converges in at most one step. When one of these two conditions does not hold, for example, true rates do not scale well due to other overheads (e.g. worker coordination) or dataflow operators have data-dependent output rates, DS2 needs more steps to converge to a stable configuration. In each of these steps, DS2 tries to minimize the error of its previous decision to get closer to the target, as any typical controller does. We omit the details of this process here and we only show empirically (in § 5.4) that DS2 needs at most three steps to converge in all our experiments. Further reducing the number of steps requires good approximation of non-linear rates, which could be gradually learned by DS2 using machine learning techniques, opening an interesting direction for future work.

4 Implementation and deployment

The DS2 controller consists of about 1500 lines of Rust running as a standalone process. Here we describe the instrumentation requirements it imposes and discuss the issues encountered integrating it with three different stream processing engines: Flink, Timely dataflow, and Heron.

4.1 Instrumentation requirements

DS2 requires a subset of the instrumentation required by bottleneck detection tools for stream processors like SnailTrail [23]. The stream processor must periodically collect and report records processed, records produced, and useful time (serialization, deserialization, processing) or waiting time per operator instance.

Flink gathers some of the metrics required by DS2 (e.g. records read and produced) by default but we extended its runtime so that each operator instance maintains local counters for (de)serialization and processing duration as well as for buffer wait time, reporting them to DS2 in configurable intervals. For record-at-a-time systems like Flink, tracking and emitting metrics for every record might incur significant overhead. Instead, we aggregate measurements per input buffer for all operators, except for sources where we aggregate per output buffer. Specifically, we have implemented a `MetricsManager` module which is responsible for gathering, aggregating, and reporting policy metrics. We assign one `MetricsManager` instance per parallel thread executing operator logic. Each

thread maintains local counters for records read, records produced, (de)serialization duration, processing duration, and waiting for input and output buffers. Source operator instances send their current local counters to the `MetricsManager` every time an output buffer gets full and regular operator instances send their local counters every time they receive a new input buffer for processing. The `MetricsManager` maintains a data structure with the current aggregate metrics of its operator instance and reports them to the outside world in configurable intervals.

Timely [32] outputs raw tracing information, which we aggregate in configurable intervals to produce metrics for DS2. We use a similar `MetricsManager`, as in Flink, which receives streams of logged events coming from Timely workers and aggregates them on the fly. Each Timely worker logs individual events of different types, such as scheduling an operator or sending a message over a data channel, along with their timestamp in nanoseconds. Recall that operator instances in Timely are not blocked on their input or output queues; instead, they are continuously spinning, i.e. they are scheduled for execution (in a round-robin fashion) even if there are no data records to process. Spinning results in a huge amount of scheduling event logs, which quickly saturate the `MetricsManager`, although most of these logs are not needed for computing the true rates. To tackle this problem, we modified Timely's logger to trace and send to the `MetricsManager` only the "useful" scheduling events, i.e. those that correspond to an operator instance doing some "useful work" for the actual computation.

Heron also by default outputs detailed, aggregated metrics [22], which are periodically collected and fed into DS2. The aggregation window depends on how frequently Heron samples its metrics and can be configured.

4.2 Integration with stream processors

DS2 is mechanism-agnostic and can be integrated with any stream processor capable of dynamically varying resources and migrating state. Figure 5 shows the high-level architecture of such an integration. Instrumented streaming jobs periodically report metrics to a repository. DS2 consists of a *Scaling Policy* component implementing the model of § 3.2, and a *Scaling Manager* monitoring the repository, invoking the policy when new metrics are available, and sending scaling commands to the stream processor.

While DS2 currently only offers scaling functionality, it could be easily extended with skew and straggler mitigation techniques as shown in Figure 5. In this case, the system would consist of multi-purpose Manager and Policy components, where the first detects the problem type (e.g., presence of skew) and the latter invokes the appropriate policy. Note that DS2 collects metrics from each operator instance separately, thus skew detection can be effortlessly implemented by the Manager.

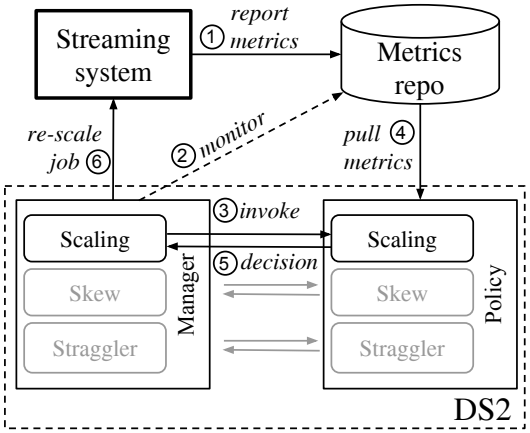


Figure 5: DS2 integration with streaming systems

We have integrated DS2 with Apache Flink, which employs a simple scaling mechanism: when instructed, Flink takes a *savepoint*, a consistent snapshot of the job state, halts the computation, and redeploys it with the updated parallelism [24]. We demonstrate this integration in action and evaluate it under a dynamic source rate in § 5.3.

4.2.1 Scaling Manager

Operational issues in real deployments that are not captured by the model must be handled by the implementation instead. To deal with factors that might affect scaling decisions in practice, the Scaling Manager provides the following configuration parameters:

Policy interval defines the frequency with which metrics are gathered and the policy invoked. Tuning the policy interval allows the scaling manager to aggregate metrics meaningfully, e.g. to ensure enough data is available to compute averages for processing and output rates. Long intervals give stable metrics but also increase reaction time. The interval must also be tuned based on the reconfiguration mechanism of the reference system. In our experiments, we found 5–30s intervals reasonable for Flink and Timely. For Heron, we found the default 60s suitable.

Warm-up time is the number of consecutive policy intervals ignored after a scaling action, since rate measurements can be unstable at the start of a computation or before backpressure builds up.

Activation time specifies when DS2 applies a scaling decision, as the number of consecutive policy decisions considered by the scaling manager before issuing a scaling command. Activation time plus an appropriate

policy interval mitigates the effects of irregularities in some streaming computations, such as non-incremental tumbling windows or data-dependent operators. For instance, consider naively-implemented window operators that buffer records and only apply the computation logic after the window fires. As long as input is simply assigned to a window, the operator's processing rate will appear high but once the window fires and the actual computation is performed the processing rate will suddenly drop. DS2 can consider several consecutive policy decisions and, for example, compute the maximum or median parallelism across intervals before applying a scaling action.

Target rate ratio defines a maximum allowed difference between the observed source rate achieved by the policy and the target rate, addressing the practical issue that processing and output rates might be affected by overheads not captured by instrumentation. For instance, adding workers to a distributed computation might incur higher coordination, channel selection cost, or resource contention, and so a computation might need more resources to achieve the target rate than the policy indicates. DS2 estimates the additional resources required by computing the ratio between the currently achieved rate and the target rate.

4.2.2 Practical considerations

DS2 also ignores minor changes (e.g. changing an operator's parallelism by one or two), which can be triggered by noisy metrics. External disruptions, such as garbage-collection in Java-based systems or disk I/O, can also influence rates measurements. For example, when integrating DS2 with Flink, we took care to properly configure task managers, heap memory, and network buffers. We are also aware that system performance might degrade after a scaling action (though we have not observed this in practice). If this were to happen, DS2 rolls back to the previous configuration. Similarly, consecutive decisions resulting in very small improvements indicate a performance issue (e.g. data skew, stragglers) that cannot be improved by scaling. DS2 can limit the number of decisions to prevent further reconfiguration.

4.2.3 DS2 in the presence of skew

Even though the scaling model assumes no data imbalance and the current implementation of DS2 does not offer skew mitigation functionality, it is worth discussing how the system behaves if skew actually appears in a streaming application it is controlling. In such a case, the system makes a scaling decision assuming data balance (§ 3.3) by averaging true processing and output rates. Thus, DS2 proposes a configuration which might not meet the target throughput but at the same time will not over-provision the system. Further, due to DS2's ability to limit the number of decisions (§ 4.2.2), the policy is guaranteed to converge. We have verified the above behavior experimentally on Flink varying the skew parameter in the Dhalion benchmark from 20% to 50% and 70%. In all cases, DS2 converged after two steps to the configuration which would be optimal if there was no skew, but which in this experiment did not meet the target throughput.

4.3 Execution model independence

DS2's policy can be applied on streaming systems regardless of their execution model. In Flink and Heron each dataflow operator is assigned a number of worker threads that define its level of parallelism, i.e. the number of parallel instances executing the operator's logic. In this case, Eq. 7 can be directly used to configure operator parallelism independently. In Timely, on the other hand, parallelism is configured globally for the whole dataflow. Each worker runs every operator in the dataflow graph according to a round-robin scheduling strategy.

For Timely, DS2 estimates the optimal number of total workers by summing up the optimal level of parallelism, as given by Eq. 7, for all operators in the dataflow. The intuition here is simple: an operator that needs π_i instances to keep up with its input actually needs $\pi_i \cdot 100\%$ computing power per unit of time. In an execution model like Timely's where operators share computing resources (worker threads), the total computing power needed so that the system can keep up with its input is $\sum_{\forall i} \pi_i \cdot 100\%$. We experimentally validate the accuracy of DS2 decisions on Timely in § 5.5.

5 Experimental evaluation

Our evaluation covers DS2 in use with three different streaming systems: Heron, Flink, and Timely Dataflow. We start our evaluation by comparing DS2 with the state-of-the-art Dhalion scaling controller used in Heron, with the benchmark in the original Dhalion publication [13]. We then demonstrate DS2 in action through end-to-end, dynamic scaling experiments with Flink, followed by measurements of DS2 convergence and accuracy in using both Flink and Timely. Finally, we evaluate the overhead of the instrumentation used by DS2.

5.1 Setup

We run all Flink and Timely experiments on up to four machines, each with 16 Intel Xeon E5-2650 @2.00GHz cores and 64GB of RAM, running Debian GNU/Linux 9.4. We use Apache Flink `1.4.1` configured with 12 TaskManagers, each with 3 slots (maximum parallelism

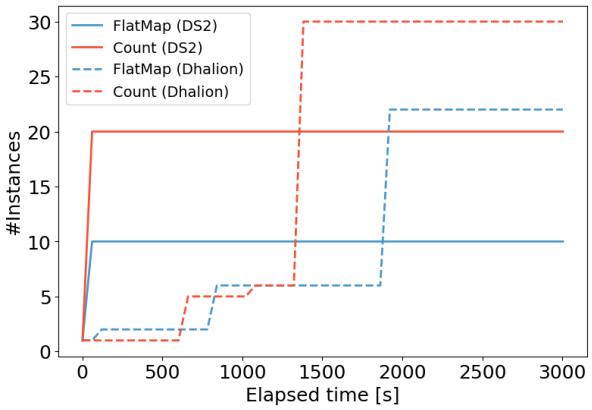


Figure 6: Comparison of DS2 vs Dhalion on Heron using the word count dataflow of [13].

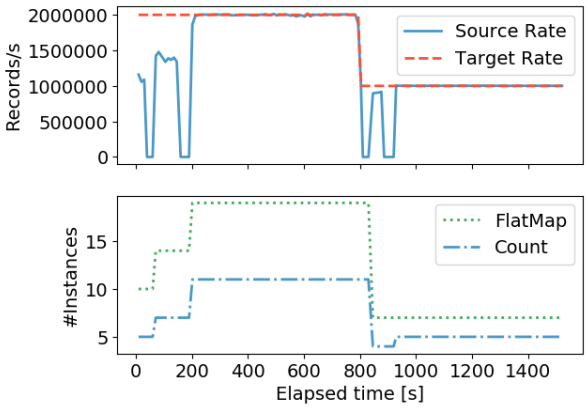


Figure 7: Dynamic scaling experiment with Flink using DS2 on the word count dataflow of [13].

per operator = 36), and Timely Dataflow `0.5.0` compiled with Rust `1.24.0`. For the comparison experiment, we run Heron `0.17.8` on a four socket-machine equipped with AMD Opteron 6276, with 64 threads in total and 256GiB of memory.

To demonstrate generality across diverse computations and streaming operators, we selected six queries from the Nexmark benchmarking suite of Apache Beam [42, 36, 37]. Specifically, we test the policy with Queries 1–3, 5, 8, and 11, which contain various representative streaming operators: stateless streaming transformations, i.e. map and filter in **Q1** and **Q2** respectively, a stateful record-at-a-time two-input operator (incremental join) in **Q3**, and various window operators: sliding window in **Q5**, tumbling window join in **Q8**, and session window in **Q11**. These queries specify computations both in processing and event time domains [5]. For the comparison with Dhalion (§ 5.2) and the end-to-end experiment on Flink (§ 5.3), we use the wordcount dataflow as specified in Dhalion's paper [13].

5.2 DS2 vs Dhalion on Heron

We compare the accuracy and convergence steps of DS2 with Dhalion, recreating the benchmark in [13].

We run Heron with Dhalion and its dynamic resource allocation policy enabled. The source operator of the three-stage wordcount topology (Source, FlatMap, Count) produces sentences at a fixed rate of 1M per minute. The FlatMap and Count operators are rate-limited to simulate bottlenecks: each FlatMap instance splits at most 100K sentences per minute, and each Count instance counts up to 1M words per minute (the same ratios as in the Dhalion paper). We start under-provisioned with one instance per operator and let Heron stabilize without backpressure.

We have already seen how the source rate evolves to match the target throughput in this experiment in Figure 1. Figure 6 shows the parallelism of FlatMap and Count over time, from the start until convergence. Dhalion makes six scale-up decisions (each involving a single operator) and reaches a stable configuration with 22 FlatMap instances and 30 Count instances after 2000 seconds.

We then apply DS2 on the same initial under-provisioned configuration using a 60*s* decision interval, no warm-up, one interval activation time, and 1.0 target ratio (cf. § 4). DS2 indicates a required parallelism of 10 for FlatMap and 20 for Count, which indeed is the minimum configuration that handles 1M sentences per minute. Note that DS2 correctly estimates the optimal parallelism in a single step, after only one minute of collecting the default Heron performance metrics.

Dhalion requires several re-configuration steps, each affecting a single operator, and reaches a final configuration that is significantly over-provisioned, even in this simple wordcount dataflow. In contrast, DS2 correctly identifies the optimal configuration in a single step and two orders of magnitude less time than Dhalion.

Besides those discussed in § 2, another reason Dhalion takes so long to reach a backpressure-free configuration is that its reaction time depends on the size of the operator queues. By default, Heron has a 100MiB buffer per operator queue, which may take some time to fill (depending on the workload) before backpressure kicks in and Dhalion can react. In contrast, DS2 only depends on the decision interval where metrics are aggregated, arbitrarily specified by the user and typically much smaller.

5.3 DS2 on Flink

We now show DS2 driving Apache Flink, in order to demonstrate the benefits of DS2 when combined with

	Bids		Auctions		Persons	
	Flink	Timely	Flink	Timely	Flink	Timely
Q1	4M	5M	—		—	
Q2	4M	5M	—		—	
Q3	—		500K	3M	100K	800K
Q5	500K	2M	—		—	
Q8	—		420K	4M	120K	4M
Q11	1M	9M	—		—	

Table 3: Target source rate (records/s) configuration for the Nexmark queries on Apache Flink and Timely.

a fast re-configuration mechanism such as that in Flink.

Here, DS2 uses a 10*s* decision interval, 30*s* warm-up time, one interval activation time, and 1.0 target ratio. DS2 hence ignores the first three decisions after re-configuration, applying a decision immediately after.

We use the same wordcount dataflow as before, this time with two phases corresponding to scale-up and scale-down scenarios respectively. In phase 1, the source rate is 2M sentences per second and Flink starts under-provisioned with 10 FlatMap instances and 5 Count instances. In this state, FlatMap can not keep up with the source rate, neither can Count handle FlatMap's output rate. Once Flink has reached a backpressure-free configuration, we keep the source rate stable for 10 minutes. During the second phase, we decrease the source rate to 1M sentences per second and keep it stable for another 10 minutes.

Figure 7 shows observed source rate and operator parallelism over time. DS2 applies two scale-up actions. First, at 40*s* it re-deploys the dataflow with 14 FlatMap instances and 7 Count instances. This happens right after the warm-up and activation time, and Flink takes around 30*s* to snapshot state and restart from the savepoint [24].

At 150*s* DS2 acts again to increase FlatMap to 19 and Count to 11 instances. This time Flink takes about 50*s* to redeploy the backpressure-free configuration at 200*s*.

At 803*s* (3*s* into the second phase) DS2 reacts to the reduced source rate by reducing the configuration to 7 FlatMap and 4 Count instances at 845*s*. At 900*s* it makes a final decision to increase Count parallelism by one, and Flink successfully applies the change at 930*s*, reaching the new optimal configuration.

This shows that DS2 plus an efficient re-configuration mechanism can offer robust dynamic scaling for streaming dataflows, allowing the reference system to react to changes in its workload in just a few seconds – significantly faster than any other systems we are aware of.

5.4 Convergence

We now show DS2 convergence from both over- and under-provisioned states on more complex dataflows. We use the same Flink configuration as before, and execute each query with fixed source rates (cf. Table 3) and initial configurations of varying parallelism. We run each query-configuration combination for 5 minutes and evaluate DS2 with 30*s* decision interval, 30*s* warm-up time, 1.0 target ratio, and five intervals activation (i.e. we consider the policy to have converged if the decision is unchanged over 5 consecutive intervals).

Table 4 shows the indicated parallelism per decision step for the main operator of each query on Flink. Note that queries **Q3**, **Q5**, **Q8**, and **Q11** include many operators, but we show results for the main operator of each for simplicity. DS2 converges in one step for simple queries and initial configurations close to optimal (e.g. **Q1** with parallelism 12), and in at most three steps for complex queries and initial configurations far from optimal (e.g. **Q5** with initial parallelism 8).

In all cases, DS2 takes at most three steps to converge. It needed three steps in 3 experiments (with **Q2**, **Q5**, and **Q11**), two steps in 14 experiments, and a single step in 19 out of 36 total experiments. We also ran the same queries using Timely Dataflow and the results were similar.

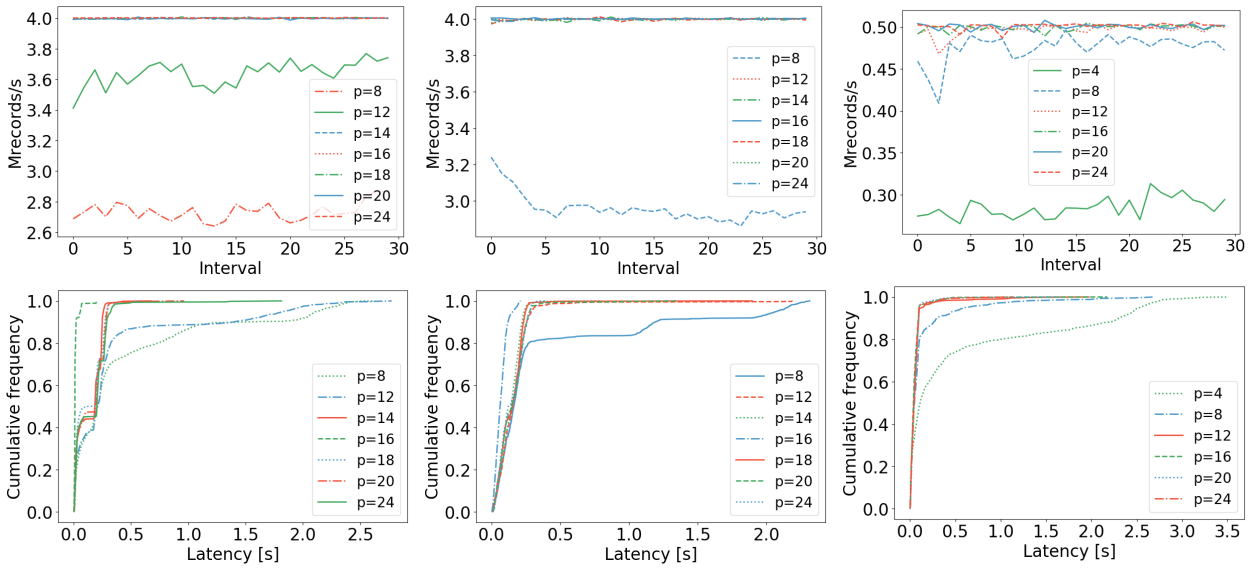
This shows that DS2 provides two important SASO properties: *stability* and *short settling time*.

Intuitively, one DS2 step moves close to optimal by estimating ideal linear scaling (§ 3.4). For far-from-optimal initial configurations, the second step "refines" this decision with a more accurate measurement, and the third step compensates for uncaptured overheads.

5.5 Accuracy

We next show accuracy: DS2 converges to configurations that exhibit no backpressure (and thus keep up with the source rates) while minimizing resource usage. In particular, we show that for a given dataflow, fixed input rate, and initial configuration, DS2 identifies the optimal parallelism regardless of whether the job is initially under- or over-provisioned. We further show that there exists no other backpressure-free configuration with lower parallelism than the one DS2 computes. Finally, we show that this configuration gives low latency by minimizing waiting time per operator instance.

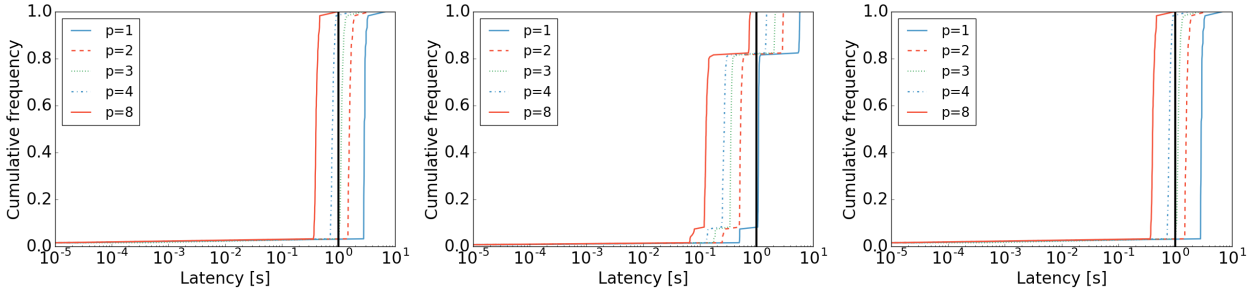
We set source rates as in Table 3 and parallelism given by the convergence experiment. Figure 8 plots observed source rates (top) and per-record latency (bottom) for the main operator of each Nexmark query on Flink with different configurations. For queries with two sources (**Q3** and **Q8**), we show results for the higher-rate source (results for the low-rate sources are similar). In all cases, DS2 successfully identifies the lowest parallelism that can keep up with the source rate. Further increasing the parallelism does not significantly improve latency and would waste resources, while lower parallelism would



(a) [Q1] Indicated parallelism: 16 (b) [Q2] Indicated parallelism: 14 (c) [Q3] Indicated parallelism: 20

(d) [Q5] Indicated parallelism: 16 (e) [Q8] Indicated parallelism: 10 (f) [Q11] Indicated parallelism: 28

Figure 8: Observed source output rates and per-record latency CDFs for different configurations of the Nexmark operators on Apache Flink.

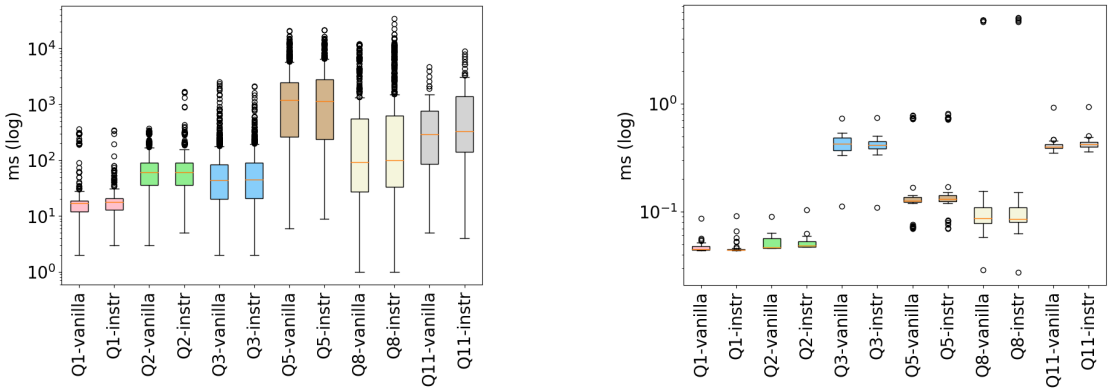


(a) [Q3] Indicated parallelism: 4 (b) [Q5] Indicated parallelism: 4 (c) [Q11] Indicated parallelism: 4

Figure 9: CDFs of per-epoch latencies for different configurations of the Nexmark operators on Timely.

Initial configuration	Q1	Q2	Q3	Q5	Q8	Q11
8	12→**16**	11→13→**14**	16→**20**	14→15→**16**	**10**	12→22→**28**
12	**16**	**14**	18→**20**	**16**	**10**	22→**28**
16	**16**	12→**14**	**20**	**16**	8→**10**	26→**28**
20	**16**	13→**14**	**20**	14→**16**	8→**10**	**28**
24	**16**	**14**	**20**	14→**16**	8→**10**	**28**
28	**16**	**14**	**20**	13→**16**	8→**10**	**28**

Table 4: DS2 convergence steps for Nexmark queries on Flink. Values are the level of parallelism of the main operator of each query. Leftmost column shows initial parallelism (from 8 to 28 instances); subsequent columns show optimal level of parallelism as estimated by DS2 in each step. Final decisions converged to by DS2 are highlighted.



(a) Flink instrumentation overhead

(b) Timely instrumentation overhead

Figure 10: Policy instrumentation overhead for the Nexmark queries of Table 3 with instrumentation disabled (`vanilla`) and enabled (`instr`) for both Flink (10a) and Timely (10b).

cause backpressure.

Timely does not have a backpressure mechanism so data sources are never delayed and the observed source rates are always equal to the initial fixed rate (instead, queues grow when the system cannot keep up). We therefore simply show CDFs of per-epoch latencies with different configurations for Timely. Figure 9 shows these for **Q3**, **Q5**, and **Q11**; results are similar for other queries. Each epoch in the CDFs corresponds to 1s of data, which must be processed in less than 1s. The optimal parallelism indicated by DS2 is $p = 4$ in all queries, regardless of the starting configuration. For **Q3** (left) and **Q11** (right), $p = 4$ is clearly the configuration that can keep up with the 1s target (vertical line in the plots) using minimum required resources. For **Q5**, 18% of the epochs are above the target by up to 0.5s. Here, the larger percentage of epochs that cannot keep up is because of the window operator, which stashes data and then forwards it at certain time points. This manifests as load spikes, which require additional resources for the system to keep up. Longer decision intervals smooth out the spikes but tend to affect policy decisions towards higher optimal configurations, which is why DS2 indicated $p = 4$ (cf. § 4.2).

In summary, DS2 identified optimal configurations in all experiments and never overshot (provisioned more resources than needed), thereby exhibiting the remaining two SASO properties: *accuracy* and *no overshoot*.

5.6 Instrumentation overhead

Finally, we evaluate instrumentation overhead. We run the Nexmark queries for 5 minutes with source rates from Table 3 and a 10s decision interval — the smallest we use in this paper, which results in the most frequently aggregated logs and has the highest potential overhead on the system performance.

We measure per-record latency in Flink using its built-in metric and per-epoch latency in Timely using 1s event-time epochs. Figure 10 shows boxplots for both systems. Individual columns show latency with logging completely off (vanilla) and instrumentation activated (instr). Overheads are small: at most 13% on Flink (40ms absolute difference) and at most 20% on Timely (5ms absolute difference) across all queries. Performance penalties are an acceptable trade-off for a good scaling policy, and could be further reduced with a larger decision interval and pre-

aggregation of metrics. Note that Heron incurs no overhead since it gathers the required metrics by default.

6 Conclusion

In this paper we have described and evaluated DS2, a novel automatic scaling controller for distributed streaming dataflows. Unlike existing scaling approaches, which rely on coarse-grained metrics and simplistic models, DS2 leverages knowledge of the dataflow graph, the computational dependencies among operators, and estimates the operators' true processing and output rates.

DS2 uses a general performance model that is mechanism-agnostic and broadly applicable to a range of streaming systems. We have implemented DS2 atop different stream processing engines: Apache Flink, Timely Dataflow, and Apache Heron, and showed that it is capable of accurate scaling decisions with fast convergence, while incurring negligible instrumentation overheads.

An interesting question for future work is what kind of scaling and adaptation mechanisms are a good match for a controller like DS2. The efficiency of DS2's model means that responsiveness is often limited by the latency of the scaling mechanism of the stream processor (when it is not determined by the granularity of measurement). All the stream processors we test against implement scaling actions by checkpointing the dataflow, redeploying, and restoring from the checkpoint. A faster, more dynamic reconfiguration mechanism might allow DS2 to operate on shorter timescales than the tens of seconds it allows in current systems.

We will release DS2 as open source, together with all code and data used to produce the results in this paper.

Acknowledgements

We thank Nicolas Hafner for his help with the Nexmark queries implementation on Timely. We would also like to thank the anonymous OSDI reviewers for their insightful comments, and our shepherd Matei Zaharia for his guidance in improving the paper. This work was partially supported by the Swiss National Science Foundation, a Google Research award, and a gift from VMware Research. Vasiliki Kalavri is supported by an ETH Postdoctoral fellowship.

References

[1] Dynamic allocation in spark. `https://www.slideshare.net/databricks/dynamic-allocation-in-spark`, 2015.

[2] Dynamic resource allocation in spark. `https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation`, 2018.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. Mcveety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[6] E. Anderson and M. Dvorsky. Comparing Cloud Dataflow autoscaling to Spark and Hadoop. `https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop`, 2016.

[7] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, pages 189–200, 2017.

[8] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.*, 10(12):1718–1729, Aug. 2017.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, 38(4), 2015.

[10] M. D. de Assunção, A. D. S. Veith, and R. Buyya. Resource elasticity for distributed data stream processing: A survey and future directions. *CoRR*, abs/1709.01363, 2017.

[11] S. Desimone. Storage reimagined for a streaming world. `http://blog.pravega.io/2017/04/09/storage-reimagined-for-a-streaming-world/`, 2017.

[12] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[13] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 2017.

[14] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

[15] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[16] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres. Dynamic configuration of partitioning in spark applications. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1891–1904, 2017.

[17] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[18] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[19] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[20] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272, 2011.

[22] Apache Heron. `https://github.com/apache/incubator-heron` (accessed: April 2018).

[23] M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. Dimitrova, S. Wicki, Z. Chothia, and T. Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, 2018. USENIX Association.

[24] F. Hueske. Savepoints: Turning Back Time. `https://data-artisans.com/blog/turning-back-time-savepoints`, 2016.

[25] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *PVLDB*, 10(11):1286–1297, 2017.

[26] E. Kirpichov and M. Denielou. No shard left behind: dynamic work rebalancing in Google Cloud Dataflow (accessed: March 2018). `https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow`.

[27] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[28] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[29] B. Lohrmann, P. Janacik, and O. Kao. Elastic Stream Processing with Latency Guarantees. In *Proceedings - International Conference on Distributed Computing Systems*, 2015.

[30] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, Dec. 2014.

[31] S. S. Manvi and G. K. Shyam. Resource management for infrastructure as a service (iaas) in cloud computing: A survey. *Journal of Network and Computer Applications*, 41:424 – 440, 2014.

[32] F. McSherry. A modular implementation of timely dataflow in Rust (accessed: April 2018). `https://github.com/frankmcsherry/timely-dataflow`.

[33] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, nov 2013.

[34] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 137–148, 2015.

[35] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 589–600, 2016.

[36] Apache Beam Nexmark benchmark suite. `https://beam.apache.org/documentation/sdks/java/nexmark`.

[37] NEXMark benchmark. `http://datalab.cs.pdx.edu/niagaraST/NEXMark`.

[38] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 233–244. IEEE, 2016.

[39] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.

[40] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, SIGMOD '14, pages 147–156, New York, New York, USA, 2014. ACM Press.

[41] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 787–798. VLDB Endowment, 2006.

[42] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Benchmark for Queries over Data Streams DRAFT. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[43] Y. Wu and K.-L. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*, pages 723–734. IEEE, apr 2015.

[44] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 22–31, 2016.

[45] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Grégory M. Essertel[1], Ruby Y. Tahboub[1], James M. Decker[1], Kevin J. Brown[2], Kunle Olukotun[2], Tiark Rompf[1]

[1]*Purdue University,* [2]*Stanford University*

{gesserte,rtahboub,decker31,tiark}@purdue.edu, {kjbrown,kunle}@stanford.edu

Abstract

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy, need to be run frequently on changing data, or interface with external libraries and systems (e.g., TensorFlow for machine learning).

We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow.

Introduction

Modern data analytics applications require a combination of different programming paradigms, spanning relational, procedural, and map-reduce-style functional processing. Systems like Apache Spark [8] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [17, 44]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this paper. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [35]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [8]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrary UDFs, but this is becoming less and less of a concern as more and more func-

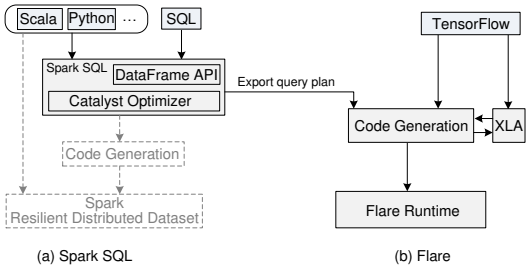


Figure 1: Flare system overview: (a) architecture of Spark adapted from [8]; (b) Flare generates code for entire queries, eliminating the RDD layer, and orchestrating parallel execution optimized for shared memory architectures. Flare also integrates with TensorFlow.

tionality is implemented on top of DataFrames.

With main-memory databases in mind, it follows that one may look to existing databases for answers on improving Spark's performance. A piece of low-hanging fruit seems to be simply translating all DataFrame query plans to an existing best-of-breed main-memory database (e.g., HyPer [35]). However, such systems are *full* database systems, not just query engines, and would require data to be stored in a separate, internal format specific to the external system. As data may be changing rapidly, loading this data into an external system is undesirable, for reasons of both storage size and due to the inherent overhead associated with data loading. Moreover, retaining the ability to interact with other systems (e.g., TensorFlow [3] for machine learning) is unclear.

Another logical alternative would be to build a new system which is overall better optimized than Spark for the particular use case of medium-size workloads and scale-up architectures. While some effort has been done in this vein (e.g., Tupleware [17]), such systems forfeit the ability to leverage existing libraries and frameworks built on top of Spark, including the associated tooling. Whereas a system that competes with Spark must replicate all of this functionality, our goal instead was to build a drop-in module capable of handling workloads for which Spark is not optimized, preferably using methodologies seen in these best-of-breed, external systems (e.g., HyPer).

Native Query Compilation Indeed, the need to accelerate CPU computation prompted the development of a code generation engine that ships with Spark since version 1.4, called Tungsten [8]. However, despite following some of the methodology set forth by HyPer, there are a number of challenges facing such a system, which causes Tungsten to yield suboptimal results by comparison. First, due to the fact that Spark resides in a Java-based ecosystem, Tungsten generates Java code. This (somewhat obviously) yields inferior performance to native execution as seen in HyPer. However, generating native code within Spark poses a challenge of interfacing with the JVM when dealing with e.g., data loading. Another challenge comes from Spark's reliance on resilient distributed datasets (RDDs) as its main internal execution abstraction. Mapping query operators to RDDs imposes boundaries between code generation regions, which incurs nontrivial runtime overhead. Finally, having a code generation engine capable of interfacing with external frameworks and libraries, particularly machine-learning oriented frameworks like TensorFlow and PyTorch, is also challenging due to the wide variety of data representations which may be used.

End-to-End Datapath Optimization In solving the problem of generating native code and working within the Java environment, we focus specifically on the issue of data processing. When working with data directly from memory, it is possible to use the Java Native Interface (JNI) and operate on raw pointers. However, when processing data directly from files, fine-grained interaction between decoding logic in Java and native code would be required, which is both cumbersome and presents high overhead. To resolve this problem, we elect to reimplement file processing for common formats in native code as well. This provides a fully compiled data path, which in turn provides significant performance benefits. While this does present a problem in calling Java UDFs (user-defined functions) at runtime, we can simply fall back to Spark's existing execution in such a case, as these instances appear rare in most use cases considered. We note in passing that existing work (e.g., Tupleware [17], Froid [40]) has presented other solutions for this problem which could be adopted within our method, as well.

Fault Tolerance on Scale-Up Architectures In addition, we must overcome the challenge of working with Spark's reliance on RDDs. For this, we propose a simple solution: when working in a scale-up, shared memory environment, remove RDDs and bypass all fault tolerance mechanisms, as they are not needed in such architectures (seen in Figure 1b). The presence of RDDs fundamentally limits the scope of query compilation to individual query stages, which prevents optimization at the granularity of full queries. Without RDDs, we compile whole queries and eliminate the preexisting boundaries across query stages. This also enables the removal of artifacts of distributed architectures, such as Spark's use of `HashJoinExchange` operators even if the query is run on a single core.

Interfacing with External Code Looking now to the issue of having a robust code generation engine capable of interfacing with external libraries and frameworks within Spark, we note that most performance-critical external frameworks are *also* embracing deferred APIs. This is particularly true for machine learning frameworks, which are based on a notion of execution graphs. This includes popular frameworks like TensorFlow [3], Caffe [24], and ONNX [1], though this list is far from exhaustive. As such, we focus on frameworks with APIs that follow this pattern. Importantly, many of these systems already have a native execution backend, which allows for speedups by generating all required glue code and keeping the entire data path within native code.

Contributions The main intellectual contribution of this paper is to demonstrate and analyze some of the underlying issues contained in the Spark runtime, and to show that the HyPer query compilation model must be adapted in certain ways to achieve good results in Spark (and, most likely, systems with a similar architecture like Flink [16]), most importantly to eliminate codegen boundaries as much as possible. For Spark, this means generating code not at the granularity of operator pipelines but compiling whole Catalyst operator trees at once (which may include multiple SQL-queries and subqueries), generating specialized code for data structures, for file loading, etc.

We present Flare, an accelerator module for Spark that solves these (and other) challenges which currently prevent Spark from achieving optimal performance on scale-up architectures for a large class of applications. Building on query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external libraries such as TensorFlow.

This paper makes the following specific contributions:

- We identify key impediments to performance for medium-sized workloads running on Spark on a single machine in a shared memory environment and present a novel code generation strategy able to overcome these impediments, including the overhead inherent in boundaries between compilation regions. (Section 2).
- We present Flare's architecture and discuss some implementation choices. We show how Flare is capable of optimizing data loading, dealing with parallel execution, as well as efficiently working on NUMA systems. This is a result of Flare compiling whole queries, as opposed to individual query stages, which results in an end-to-end optimized data path (Section 3).
- We show how Flare's compilation model efficiently extends to external user-defined functions. Specifically, we discuss Flare's ability to integrate with other frameworks and domain-specific languages, including in particular machine learning frameworks like TensorFlow that provide compilation facilities of their own (Section 4).
- We evaluate Flare in comparison to Spark on TPC-H, reducing the gap to best-of-breed relational query engine, and on benchmarks involving external libraries. In both settings, Flare exhibits order-of-magnitude speedups. Our evaluation spans single-core, multi-core, and NUMA targets (Section 5).

Finally, we survey related work in Section 6, and draw conclusions in Section 7.

Background

Apache Spark [55, 56] is today's most widely-used big data framework. The core programming abstraction comes in the form of an immutable, implicitly distributed collection called a resilient distributed dataset (RDD). RDDs serve as high-level programming interfaces, while also transparently managing fault-tolerance.

We present a short example using RDDs (from [8]), which counts the number of errors in a (potentially distributed) log file:

```
val lines  = spark.sparkContext.textFile("...")
val errors = lines.filter(s => s.startsWith("ERROR"))
println("Total errors: " + errors.count())
```

Spark's RDD abstraction provides a *deferred* API: in the above example, the calls to `textFile` and `filter` merely construct a computation graph. In fact, no actual computation occurs until `errors.count` is invoked.

The directed, acyclic computation graph represented by an RDD describes the distributed operations in a rather coarse-grained fashion: at the granularity of `map`, `filter`, etc. While this level of detail is enough to enable demand-driven computation, scheduling, and fault-tolerance via selective recomputation along the "lineage" of a result [55], it does not provide a full view of the computation applied to each element of a dataset. For example, in the code snippet shown above, the argument to `lines.filter` is a normal Scala closure. This makes integration between RDDs and arbitrary external libraries much easier, but it also means that the given closure must be invoked as-is for every element in the dataset.

As such, the performance of RDDs suffers from two limitations: first, limited visibility for analysis and optimization (especially standard optimizations, e.g., join

reordering for relational workloads); and second, substantial interpretive overhead, i.e., function calls for each processed tuple. Both issues have been ameliorated with the introduction of the Spark SQL subsystem [8].

The Power of Multi-Stage APIs

The chief addition of Spark SQL is an alternative API based on DataFrames. A DataFrame is conceptually equivalent to a table in a relational database; i.e., a collection of rows with named columns. However, like RDDs, the DataFrame API records operations, rather than computing the result.

Therefore, we can write the same example as before:

```
val lines  = spark.read.textFile("...")
val errors = lines.filter($"value".startsWith("ERROR"))
println("Total errors: " + errors.count())
```

Indeed, this is quite similar to the RDD API in that only the call to `errors.count` will trigger actual execution. Unlike RDDs, however, DataFrames capture the *full* computation/query to be executed. We can obtain the internal representation using `errors.explain()`, which produces the following output:

```
== Physical Plan ==
*Filter StartsWith(value#894, ERROR)
+- *Scan text [value#894]
     Format: ...TextFileFormat@18edbdbb,
     InputPaths: ...,
     ReadSchema: struct<value:string>
```

From the high-level DataFrame operations, Spark SQL computes a *query plan*, much like a relational DBMS. Spark SQL optimizes query plans using its relational query optimizer, called Catalyst, and may even generate Java code at runtime to accelerate parts of the query plan using a component named Tungsten (see Section 2.2).

It is hard to overstate the benefits of this kind of deferred API, which generates a complete program (i.e., query) representation at runtime. First, it enables various kinds of optimizations, including classic relational query optimizations. Second, one can use this API from multiple front-ends, which exposes Spark to non-JVM languages such as Python and R, and the API can also serve as a translation target from literal SQL:

```
lines.createOrReplaceTempView("lines")
val errors = spark.sql("select * from lines
                        where value like 'ERROR%'")
println("Total errors: " + errors.count())
```

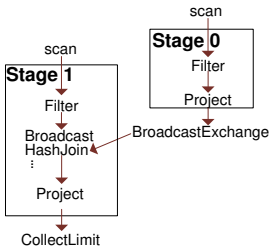
Third, one can use the full host language to structure code, and use small functions that pass DataFrames between them to build up a logical plan that is then optimized as a whole.

However, this is only true as long as one stays in the relational world, and, notably, avoids using any external libraries (e.g., TensorFlow). This is a nontrivial restriction; to resolve this, we show in Section 4 how the DataFrame model extends to such library calls in Flare.

```
select *
from lineitem, orders
where l_orderkey = o_orderkey

Spark
  Sort-merge join         14,937 ms
  Broadcast-hash join      4,775 ms
    of which in exchange 2,232 ms
Flare
  In-memory hash join       136 ms
```

(a)



(b)

Figure 2: (a) The cost of Join `lineitem` ⋈ `orders` with different operators (b) Spark's hash join plan shows two separate code generation regions, which communicate through Spark's runtime system.

Catalyst and Tungsten

With the addition of Spark SQL, Spark also introduced a query optimizer known as Catalyst [8]. We elide the details of Catalyst's optimization strategy, as they are largely irrelevant here. After Catalyst has finished optimizing a query plan, Spark's execution backend known as Tungsten takes over. Tungsten aims to improve Spark's performance by reducing the allocation of objects on the Java Virtual Machine (JVM) heap, controlling off-heap memory management, employing cache-aware data structures, and generating Java code which is then compiled to JVM bytecode at runtime [28]. Notably, these optimizations are able to simultaneously improve the performance of *all* Spark SQL libraries and DataFrame operations [56].

Following the design described by Neumann, and implemented in HyPer [35], Tungsten's code generation engine implements what is known as a "data-centric" model. In this type of model, operator interfaces consist of two methods: `produce`, and `consume`. The `produce` method on an operator signals all child operators to begin producing data in the order defined by the parent operator's semantics. The `consume` method waits to receive and process this data, again in accordance with the parent operator's semantics.

In HyPer (and Tungsten), operators that materialize data (e.g., aggregate, hash join, etc.) are called "pipeline breakers". Where possible, pipelines of operators (e.g., scan, aggregate) are fused to eliminate unnecessary function calls which would otherwise move data between operators. A consequence of this is that all code generated is at the granularity of query *stage*, rather than generating code for the query as a whole. This requires some amount of "glue code" to also be generated, in order to pipeline these generated stages together. The directed graph of the physical plan for a simple join query can be seen in Figure 2b. In this figure, we can see that the first stage generates code for scanning and filtering the first

```
val tpchq6 = spark.sql("""
  select
    sum(l_extendedprice*l_discount) as
          revenue
  from
    lineitem
  where
    l_shipdate >= to_date('1994-01-01')
    and l_shipdate < to_date('1995-01-01')
    and l_discount between 0.05 and 0.07
    and l_quantity < 24
""")
```

(a)

```
// data loading elided ...
for (i = 0; i < size; i++) {
  double l_quantity = l_quantity_col[i];
  double l_extendedprice = l_extendedprice_col[i];
  double l_discount = l_discount_col[i];
  long l_shipdate = l_shipdate_col[i];
  if (l_shipdate >= 19940101L && l_shipdate <
      19950101L &&
      l_discount >= 0.05 && l_discount <= 0.07 &&
      l_quantity < 24.0) {
    revenue += l_extendedprice * l_discount;
  }
} ...
```

(b)

Spark SQL	Preload ms	Query ms
Direct CSV	-	24,400
Preload CSV	118,062	1,418
Hand-Written C / Flare		
Preload CSV	2,847	45

(c)

Figure 3: (a) Query 6 from the TPC-H benchmark in Spark (b) Q6 hand-written C code (c) Running times for Q6 in Spark, with and without pre-loading, and compared to hand-written code and Flare.

table and the second stage generates code for the pipeline of the scan, join, and project operators. In Section 2.4 we discuss the impact of the granularity of code generation and the choice of join algorithm on Spark's performance.

Spark Performance Analysis

Spark performance studies primarily focus on the scale-out performance, e.g., running big data benchmarks [55] on high-end clusters, performing terabyte sorting [56], etc. However, when considering the class of computationally-heavy workloads that can fit in main-memory, requires multiple iterations, or integrates with external libraries (e.g., training a machine learning classifier), the performance of Spark becomes suboptimal.

On a similar note, McSherry, Isard, and Murray have eloquently argued in their 2015 HotOS paper [30] and accompanying blog post [29] that big data systems such as Spark tend to scale well, but often this is because there is a lot of internal overhead. In particular, McSherry et al. demonstrate that a straightforward native implementation of the PageRank algorithm [37] running on a single laptop can outperform a Spark cluster with 128 cores, using the then-current version.

Laptop vs. Cluster Inspired by this setup and the following quote, we are interested in gauging the inherent overheads of Spark and Spark SQL in absolute terms:

> "You can have a second computer once you've shown you know how to use the first one."
>
> — Paul Barham, via [30]

For our benchmark, we pick the simplest query from the industry-standard TPC-H benchmark: Query 6 (shown in Figure 3a). We define the schema of table `lineitem`, provide the source file, and finally register it as a temporary table for Spark SQL (steps not shown). For our experiments, we use scale factor 2 (SF2) of the TPC-H data set, which means that table `lineitem` is stored in a CSV file of about 1.4 GB. Following the setup by McSherry et al., we run our tests on a fairly standard laptop.[1] All times referenced below may be found in Figure 3c.

We first do a naive run of our query, Q6. As reported in Figure 3, we achieve a result of 24 seconds, which is clearly suboptimal. In aiming to boost performance, one option at this is to convert our data to the columnar Parquet format [7] for increased performance. Alternatively, we can preload the data so that subsequent runs are purely in-memory. As we are mainly interested in the computational part, we opt to preload.

We note in passing that preloading is quite slow (almost 2 min), which may be due to a variety of factors. With things preloaded, however, we can now execute our query in-memory, and we get a much better result of around 1.4 seconds. Running the query a few more times yields further speedups, but timings stagnate at around 1 second (timing from subsequent runs elided). Using 1s as our baseline, we must now qualify this result.

Hand-Written C Due to the simplicity of Q6, we elect to write a program in C which performs precisely the same computation: mapping the input file into memory using the `mmap` system call, loading the data into an in-memory columnar representation, and then executing the main query loop (see Figure 3b).

Compiling this C program via `gcc -O3 Q6.c` and running the resultant output file yields a time of 2.8 seconds (including data loading), only 45ms of which is performing the actual query computation. Note that in comparison to Spark 2.0, this is a striking 20× speedup. Performing the same query in HyPer, however, takes only 46.58ms, well within the margin of error of the hand-written C code. This disparity in performance shows that although Tungsten is written with the methodologies prescribed by HyPer in mind, there exist some impediments either in the implementation of these methodologies or in the Spark runtime itself which prevent Spark from achieving optimal performance for these cases.

[1] MacBook Pro Retina 2012, 2.6 GHz Intel Core i7, 16 GB 1600 MHz DDR3, 500 GB SSD, Spark 2.0, Java HotSpot VM 1.8.0_112-b16

```
case class BroadcastHashJoinExec(/* ... inputs elided ... */)
  extends BinaryExecNode with HashJoin with CodegenSupport {
  // ... fields elided ...
  override def doProduce(ctx: CodegenContext): String =
    streamedPlan.asInstanceOf[CodegenSupport].produce(ctx, this)
  override def doConsume(ctx: CodegenContext, input:
        Seq[ExprCode], row: ExprCode): String = {
    val (broadcastRelation, relationTerm) = prepareBroadcast(ctx)
    val (keyEv, anyNull) = genStreamSideJoinKey(ctx, input)
    val (matched, checkCondition, buildVars) =
          getJoinCondition(ctx, input)
    val numOutput = metricTerm(ctx, "numOutputRows")

    val resultVars = ...
    ctx.copyResult = true
    val matches = ctx.freshName("matches")
    val iteratorCls = classOf[Iterator[UnsafeRow]].getName
    s"""
      |// generate join key for stream side
      |${keyEv.code}
      |// find matches from HashRelation
      |$iteratorCls $matches = $anyNull ? null :
            ($iteratorCls)$relationTerm.get(${keyEv.value});
      |if ($matches == null) continue;
      |while ($matches.hasNext()) {
      |  UnsafeRow $matched = (UnsafeRow) $matches.next();
      |  $checkCondition
      |  $numOutput.add(1);
      |  ${consume(ctx, resultVars)}
      |}
    """.stripMargin
  }
}
```

Figure 4: Spark implementation of inner HashJoin.

Major Bottlenecks

By profiling Spark SQL during a run of Q6, we are able to determine two key reasons for the large gap in performance between Spark and HyPer. Note that while we focus our discussion mainly on Q6, which requires low computational power and uses only trivial query operators, these bottlenecks appear in nearly every query in the TPC-H benchmark.

Data Exchange Between Code Boundaries We first observe that Tungsten must generate multiple pieces of code: one for the main query loop, the other an iterator to traverse the in-memory data structure.

Consider the HashJoin code in Figure 4. We can see that Tungsten's produce/consume interface generates a loop which iterates over data through an iterator interface, then invokes the `consume` method at the end of the loop in order to perform evaluation. HyPer's original codegen model is centrally designed around data-centric pipelines within a given query, the notion of "pipeline-breakers" as coarse-grained boundaries of data flow, and the combination of pre-written code at the boundary between pipelines with generated code within each pipeline. While the particular implementation of this design in HyPer leads to good results in HyPer itself, the direct implementation of HyPer's pipeline-focused approach in Spark and similar systems falls short because the overhead of traversing pipeline boundaries is much higher (Java vs C++, RDD overhead, ecosystem integration, etc).

The CPU profile (Figure 5) shows that 80% of the execution time is spent in one of two ways: accessing and decoding the in-memory data representation, or moving between the two pieces of generated code through code paths which are part of the precompiled Spark runtime. In order to avoid this overhead, then, we must replace the runtime altogether with one able to reason about the *entire* query, rather than just the stages.

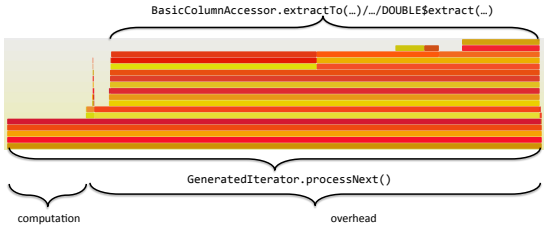


Figure 5: CPU profile of TPC-H Q6 in Spark SQL, after preloading the `lineitem` table. 80% of time is spent accessing and decoding the in-memory data representation.

JVM Overhead Even if the previous indirection is removed and replaced with a unified piece of Java code, the performance remains approximately 30% lower than our hand-written C code. This difference becomes more pronounced in other TPC-H queries which require both memory management and tighter low-level control over data structures. This bottleneck is certainly expected, and choosing a lower level language does alleviate this performance loss greatly.

Other Bottlenecks As shown, even fixing these bottlenecks is not enough. This becomes even more apparent when moving away from Q6. In dealing with more complex queries, concerns regarding granularity of code generation and the necessity to interface with the Spark runtime system become more pronounced than with TPC-H Q6. In fact, queries which require join operations exhibit some unfortunate consequences for main-memory execution due to Spark's design as primarily a cluster-computing framework. Figure 2a shows timings for a simple join query that joins the `lineitem` and `orders` tables of the TPC-H benchmark. Spark's query optimizer picks an expensive sort-merge join by default. Note that this may be the correct choice for distributed or out-of-core execution, but is suboptimal for main memory. With some tuning, it is possible to force Spark's query planner to opt for a hash join instead, which is more efficient for our architecture. However, even this follows a broadcast model with high overhead for the internal exchange operator (2.2s of 4.7s) which is present in the physical plan even when running on a single core.

Flare: Adding Fuel to the Fire

Based on the observations made in Sections 2.3 and 2.4, we formally present Flare: a new backend which acts as an accelerator for Spark for medium-sized workloads

on scale-up architectures. Flare eliminates all previously identified bottlenecks *without* removing the expressiveness and power of its front-ends. At its core, Flare efficiently generates code, and brings Spark's performance closer to HyPer and hand-written C. Flare compiles whole queries instead of only query *stages*, effectively bypassing Spark's RDD layer and runtime for operations like hash joins in shared-memory environments. Flare also goes beyond purely relational workloads by adding another intermediate layer between query plans and generated code.

As previously identified, this need to build a new runtime, rather than selecting an existing system as an alternate backend for Spark, is founded on a number of justifications. In particular, we focus on the deferred API provided by Spark SQL which builds computation graphs to perform the necessary queries as given by users. Access to this graph structure allows for cross-optimization with external libraries also using deferred APIs (e.g., TensorFlow) through the use of robust code generation techniques. In order to gain access to the necessary data at the appropriate time without incurring the overhead of passing (potentially large amounts of) data between external programs, a new runtime capable of interfacing with Spark's existing front-end is required.

Interface between Spark and Flare

Flare supports most available Spark SQL DataFrame or DataSet operations (i.e., all operations which can be applied to a DataFrame and have a representation as Catalyst operators), though any operators currently missing could be added without compatibility constraints. In the event that a Spark job contains operations that are not part of the SQL frontend, Flare can still be used to accelerate SQL operations and then return the result to the Spark runtime, which will then use the result for the rest of the computation. However, the benefit of doing this may be negated by the communication overhead between the two systems.

Flare can operate in one of two modes. Either users must invoke a function to convert the DataFrame they wish to compute into a Flare DataFrame (a conversion that may fail with a descriptive error), to that end Flare exposes a dedicated API to allow users to pick which DataFrames to evaluate through Flare:

```
val df = spark.sql("...") // create DataFrame (SQL or direct)
val fd = flare(df)        // turn it into a FlareDataFrame
fd.show()                 // execute query plan with Flare
```

Or one can set a configuration item in Spark to use Flare on all queries where possible, and only fall back to the default Spark execution when necessary (optionally emitting a warning when doing so). When Flare is

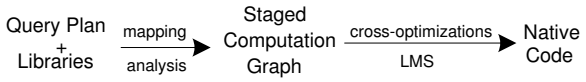


Figure 6: Query compilation in Flare

invoked, it first generates C code as explained in the following section, then invokes a C compiler, and finally launches the resulting binary either inside the JVM, or as a separate process. This bypasses Spark's runtime entirely, relying solely on Flare's runtime to trigger execution of the generated code.

Flare: Architecture

A high-level view of Flare's architecture is illustrated in Figure 1b. Spark SQL's front-end, the DataFrame API, and the Catalyst optimizer all remain the same. When dealing with relational workloads, the optimized query plan is exported without modification from Catalyst to Flare, upon which Flare performs a compilation pass and creates a code generation object for each of the nodes.

At a closer look, Figure 6 illustrates an end-to-end execution path in Flare. Flare analyzes Spark's optimized plan (which possibly embeds external libraries as UDFs) and constructs a computation graph that encodes relational operators, data structures, UDFs, data layout, and any needed configurations.

Generative Programming and Lightweight Modular Staging (LMS) Flare uses Lightweight Modular Staging (LMS) for code generation due to its multi-staging capabilities. In LMS, a special type constructor `Rep[T]` is used to denote a *staged* expression, which will cause an expression of type `T` to be emitted in the generated code. The example in Figure 7a shows a code snippet that generates a for loop (notice the loop counter is specialized with `Rep`). At evaluation time, the for loop iteration will be generated. On the other hand, the `if` condition is composed of a regular Boolean type, so this code is executed at code generation time as shown in Figure 7b.

```
val squared: Boolean = true
val arr = NewArray[Int](5)
for (i <- 0 until 5: Rep[Range]) {
  if (squared)
    arr(i) = i * i
  else
    arr(i) = i
}
```

(a)

```
int* x1 = malloc(5 *
     sizeof(int));
int x2;
for (x2 = 0; x2 < 5; x2++) {
 int x3 = x2 * x2;
 x1[x2] = x3;
}
```

(b)

Figure 7: (a) Generic LMS code example (only used for Flare internals) (b) LMS generated C code.

The essence of multi-stage programming is to generate efficient programs using high-level constructs without incurring runtime overheads [47]. In the late 1990s, it was realized that multi-stage languages (i.e., languages used

to express multi-stage programs) are useful not only as intermediate formal description, but also directly as programming tools, giving rise to the field of programmable specialization or *generative programming*, embodied by languages like MetaML [48] and MetaOCaml [15]. Even more recently, library-based approaches have become popular that implement generative programming abstractions using operator overloading and other features in regular general-purpose languages. One instance of such a system is LMS [43], implemented in Scala. LMS maintains a graph-like intermediate representation (IR) to encode constructs and operations. LMS introduces a type constructor called `Rep[T]` (where T is a type, e.g., `String`) to denote expressions that will generate code. For example, given two `Rep[Int]` values `a` and `b`, *evaluating* the expression `a+b` will *generate* the following code: `int x1 = a + b` and return a reference to `x1` as new `Rep[Int]` result in the meta language. This value can then be used in other computations.

Staging and code generation in Flare In the context of query compilation, LMS is used to specialize a query evaluator with respect to a query plan [42, 25]. Based on partial evaluation results (the first Futamura projection [21]), the outcome of programmatic specialization is a compiled target of the query. Figure 8 shows an example of compiling a join query in Flare, in which the specialization logic (i.e., staging code using `Rep`) is placed at the granularity of low-level control flow constructs and primitive operators.

```
case class DataLoop(foreach: (Rep[Record] => Unit) => Unit)
type ThreadCallback = Rep[Int] => DataLoop => Unit
case class CodeGen(gen: ThreadCallback => Unit)

// extract the join hash key functions
def compileCond(cond: Option[Expression]): (Rep[Record] =>
      Rep[Record], Rep[Record] => Rep[Record]) = ...
def compile(plan: LogicalPlan): CodeGen = plan match {
  case HashJoin(left, right, Inner, cond) =>
    val lGen = compile(left); val rGen = compile(right)
    val (lkey, rkey) = compileCond(cond)

    val hmap = new ParHashMap[Record,Record]()
    CodeGen(threadCallback =>
      lGen.gen { tId => dataloop => // start section for left child
        val lhmap = hmap.partition(tId) // Thread local data
        for (ltuple <- dataloop) lhmap += (lkey(ltuple), ltuple)
      }
      rGen.gen { tId => dataloop => // start section for right
            child
        threadCallback(tId) { callback => // invoke downstream op
          for (rtuple <- dataloop)
            for (ltuple <- hmap(rkey(rtuple)))
              callback(merge(ltuple, rtuple)) // feed downstream op
    } } )
  case ...
}
```

Figure 8: Internal Flare operator that generates code for HashJoin (LogicalPlan and HashJoin are Spark classes).

Following the `InnerJoin` code generation example in Figure 8, a `CodeGen` object is generated from each of the two children, after which the logic of the `Join` operator is implemented: the left child's code generator is invoked and the tuples produced populate a hash map. The right child's code generator is then invoked, and for each of the tuples produced, the matching lines from the left table are extracted from the map, merged, and finally become the produced value of the `Join` operator. LMS performs some lightweight optimizations (e.g., common subexpression elimination, dead code elimination), and generates C code that can be compiled and executed by the Flare runtime.

Interestingly, this implementation looks exactly like the implementation of an interpreter. Indeed, this is no coincidence: much like Spark uses multi-stage APIs (Section 2.1) at the operational level, Flare uses the LMS compiler framework, which implements the same concept, but at a lower level. In the same way that Scala (or Python) is used to build DataFrames in Spark, we use Scala to build a graph which represents the computations needing to be generated. We qualify the code generation of Spark as coarse-grain. The `BroadcastHashJoinExec` operator in Figure 4 generates a string that corresponds to the full join computation. This String is generated with regard to some placeholders for the inputs/outputs and join conditons that are specific to the given query. However, what is hardcoded in the template string will be generated in the same way for every join. Contrast this with Flare's fine-grained code generation: The code in Figure 8 also generates code for the Join operator. However, it does not generate one big string; rather, it invokes functions that express the logic of the operator using the full power of the Scala language. The use of `Rep[T]` expressions in judicious places triggers code generation and produces only low-level operations.

With the goal of removing the tightest bottlenecks first, the implementation of Flare has focused on maximizing performance within a single machine. Therefore, Flare does not implement any specific optimizations for distributed execution. Furthermore, Flare is also unable to handle any workloads which require more memory than the machine has available. In either of these cases, we fall back to the Spark runtime.

Optimizing Data Loading

Data loading is an often overlooked factor data processing, and is seldom reported in benchmarks. However, we recognize that data loading from CSV can often be the dominant performance factor for Spark SQL queries. The Apache Parquet [7] format is an attractive alternative, modeled after Dremel [31]. As a binary columnar format, it offers opportunities for compression, and queries can load only required columns.

While Parquet allows for irrelevant data to be ignored almost entirely, Spark's code to read Parquet files is very generic, resulting in undue overhead. This generality is

primarily due to supporting multiple compression and encoding techniques, but there also exists overhead in determining which column iterators are needed. While these sources of overhead seem somewhat unavoidable, in reality they can be resolved by generating specialized code. In Flare, we implement compiled CSV and Parquet readers that generate native code specialized to a given schema. As a result, Flare can compile data paths end-to-end. We evaluate these readers in Section 5.

Indexing Structures

Query engines build indexing structures to minimize time spent in table lookups to speed-up query execution. Small-size data processing is performed efficiently using table scans, whereas very large datasets are executed in latency-insensitive contexts. On the other hand, medium-size workloads can profit from indexes, as these datasets are often processed under tight latency constraints where performing full table scans is infeasible. On that basis, Flare supports indexing structures on primary and foreign keys. At the time of writing, Spark SQL does not support index-based plans. Thus, Flare adds metadata to the table schema that describes index type and key attributes. At loading time, Flare builds indexes as specified in the table definition. Furthermore, Flare implements a set of index-based operators, e.g., scan and join following the methodology described in [49]. Finally, at compilation time, Flare maps Spark's operators to use the index-based operators if such an index is present. The index-based operators are implemented with the same technique described for the basic operators, but shortcut some computation by using the index rather than requesting data from its children.

Parallel and NUMA Execution

Query engines can implement parallelism either explicitly through special *split* and *merge* operators, or internally by modifying the operator's internal logic to orchestrate parallel execution. Flare does the latter, and currently realizes parallelism using OpenMP [2] annotations within the generated C code, although alternatives are possible. On the architectural level, Flare handles splitting the computation internally across multiple threads, accumulating final results, etc. For instance, the parallel `scan` starts a parallel section, which sets the number of threads and invokes the downstream operators in parallel through a `ThreadCallback` (see Figure 8). `join` and `aggregate` operators, in turn, which implement materialization points, implement their `ThreadCallback` method in such a way that parallel invocations are possible without conflict. This is typically accomplished through either per-thread data structures that are merged after the parallel section or lock-free data structures.

Flare also contains specific optimizations for environments with non-uniform memory access (NUMA), including pinning threads to specific cores and optimizing the memory layout of various data structures to reduce the need for accessing non-local memory. For instance, memory-bound workloads (e.g., TPC-H Q6) perform small amounts of computation, and do not scale up given a large number of threads on a single CPU socket. Flare's code generation supports such workloads through various data partitioning strategies in order to maximize local processing and to reduce the need for threads to access non-local memory as illustrated Section 5.1.

Heterogeneous Workloads

Many data analytics applications require a combination of different programming paradigms, e.g., relational, procedural, and map-reduce-style functional processing. For example, a machine learning (ML) application might use relational APIs for the extract, transform, load phase (ETL), and dedicated ML libraries for computations. Spark provides specialized libraries (e.g., ML pipelines), and supports user-defined functions to support domain-specific applications. Unfortunately, Spark's performance is greatly diminished once DataFrame operations are interleaved with calls to external libraries. Currently, Spark SQL optimization and code generation treat calls to such libraries as calls to black boxes. Hence, Flare focuses on generating efficient code for heterogeneous workloads including external systems e.g., TensorFlow [4].

User Defined Functions (UDF)

Spark SQL uses Scala functions, which appear as a black box to the optimizer. As mentioned in Section 3.2, Flare's internal code generation logic is based on LMS, which allows for multi-stage programming using `Rep` types. Extending UDF support to Flare is achieved by injecting `Rep[A] => Rep[B]` functions into DataFrames in the same way that normal `A => B` functions are injected in plain Spark. As an example, here is a UDF `sqr` that squares a given number:

```
// define and register UDF
def sqr(fc: FlareUDFContext) = { import fc._;
    (y: Rep[Int]) => y * y }
flare.udf.register("sqr", sqr)
// use UDF in query
val df = spark.sql("select ps_availqty from partsupp where
                    sqr(ps_availqty) > 100")
flare(df).show()
```

Notice that the definition of `sqr` uses an additional argument of type `FlareUDFContext`, from which we import overloaded operators such as `+`, `-`, `*`, etc., to work on `Rep[Int]` and other `Rep[T]` types. The staged function will become part of the code as well, and will be

```
# Define linear classifier using TensorFlow
import tensorflow as tf
# weights from pre-trained model elided
mat = tf.constant([[...]])
bias = tf.constant([...])
def classifier(c1,c2,c3,c4):
  # compute distance
  x = tf.constant([[c1,c2,c3,c4]])
  y = tf.matmul(x, mat) + bias
  y1 = tf.session.run(y1)[0]
  return max(y1)
# Register classifier as UDF: dumps TensorFlow graph to
# a .pbtxt file, runs tf_compile to obtain .o binary file
flare.udf.register_tfcompile("classifier", classifier)
# Use compiled classifer in PySpark query with Flare:
q = spark.sql("
  select real_class,
    sum(case when class = 0 then 1 else 0 end) as class1,
    sum(case when class = 1 then 1 else 0 end) as class2,
    ... until 4 ...
  from (select real_class,
               classifier(c1,c2,c3,c4) as class from data)
  group by real_class order by real_class")
flare(q).show()
```

Figure 9: Spark query using TensorFlow classifier as a UDF in Python.

optimized along with the relational operations. This provides benefits for UDFs (general purpose code embedded in queries), and enables queries to be be optimized with respect to their surrounding code (e.g., queries run within a loop).

Native UDF Example: TensorFlow

Flare has the potential to provide significant performance gains with other machine learning frameworks that generate native code. Figure 9 shows a PySpark SQL query which uses a UDF implemented in TensorFlow [3, 4]. This UDF performs classification via machine learning over the data, based on a pretrained model. It is important to reiterate that this UDF is seen as a black box by Spark, though in this case, it is also opaque to Flare.

Calling TensorFlow code from Spark hits a number of bottlenecks, resulting in poor performance (see Section 5). This is in large part due to the separate nature of the two programs; there is no inherent way to "share" data without copying back and forth. A somewhat immediate solution is to use the JNI, which enables the use of TensorFlow's ahead-of-time (AOT) compiler, XLA [50]. This already improves performance by over 100×, but even here there is room for improvement.

Using Flare in conjunction with TensorFlow provides speedups of over 1,000,000× when compared with Spark (for concrete numbers, see Section 5). These gains come primarily as a result of Flare's ability to link with external C libraries. As mentioned previously, in this example, Flare is able to take advantage of XLA, whereas Spark is relegated to using TensorFlow's less efficient dynamic runtime (which executes a TensorFlow computation graph with only limited knowledge). Flare provides a function `flare.udf.register_tfcompile`, which internally creates a TensorFlow subgraph representing the UDF, saves it to a file, and then invokes TensorFlow's AOT compiler tool `tfcompile` to obtain a compiled object file, which can then be linked against the query code generated by Flare.

Finally, the TensorFlow UDF generated by XLA is pure code, i.e., it does not allocate its own memory. Instead, the caller needs to preallocate all memory which will be used by the UDF. Due to its ability to generate native code, Flare can organize its own data structures to meet TensorFlow's data requirements, and thus does not require data layout modification or extraneous copies.

Experimental Evaluation

To assess the performance and acceleration potential of Flare in comparison to Spark, we present two sets of experiments. The first set focuses on a standard relational benchmark; the second set evaluates heterogeneous workloads, consisting of relational processing combined with a TensorFlow machine learning kernel. Our experiments span single-core, multi-core, and NUMA targets.

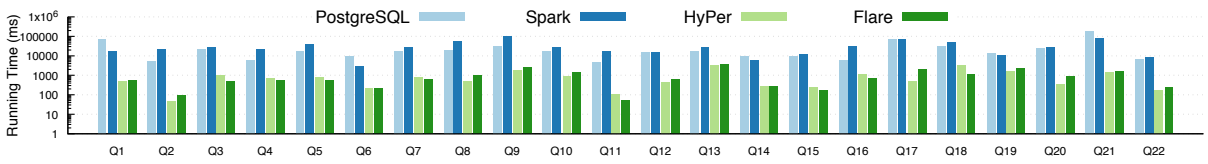
Bare-Metal Relational Workloads

The first set of experiments focuses on a standard relational workload, and demonstrates that the inherent overheads of Spark SQL cause a slowdown of at least 10× compared to the best available query engines for in-memory execution on a single core. Our experiments show that Flare is able to bridge this gap, accelerating Spark SQL to the same level of performance as state-of-the-art query compiler systems, while retaining the flexibility of Spark's DataFrame API. We also compare parallel speedups, the effect of NUMA optimization, and evaluate the performance benefits of optimized data loading.

Environment We conducted our experiments on a single NUMA machine with 4 sockets, 24 Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu 16.04.4 LTS. We use Spark 2.3, Scala 2.11, Postgres 10.2, HyPer v0.5-222-g04766a1, and GCC 5.4 with optimization flags `-O3`.

Dataset We use the standard TPC-H [51] benchmark with scale factor SF10 for sequential, and SF20 and SF100 for parallel execution.

Single-Core Running Time In this experiment, we compare the single-core, absolute running time of Flare with Postgres, HyPer, and Spark using the TPC-H benchmark with scale factor SF10. In the case of Spark, we use a single executor thread, though the JVM may spawn auxiliary threads to handle GC or the just-in-time compilation. Postgres and HyPer implement cost-based optimizers that can avoid inefficient query plans, in partic-



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	72599	5064	20790	6032	17904	9071	16408	20003	31728	16409	4729	15755	16152	9369	9607	5662	66254	30104	12852	22795	184736	6900
Spark SQL	17692	21062	28039	20693	40008	2826	28262	55346	98216	27585	16193	15431	25632	5913	11538	30973	66089	47542	11062	27448	78896	8687
HyPer	498	47	969	725	821	207	804	467	1782	846	111	460	3307	283	227	1112	485	3061	1666	345	1349	177
Flare	550	98	481	544	527	212	596	1016	2688	1429	53	643	3499	256	175	679	1924	1150	2236	836	1533	241

Figure 10: Performance comparison of Postgres, HyPer, Spark SQL, Flare in SF10

ular by reordering joins. While Spark's Catalyst optimizer [8] is also cost-based, the default configurations do not perform any kind of join re-ordering. Hence, we match the join ordering of the query plan in Spark SQL and Flare with HyPer's, with a small number of exceptions: in Spark SQL, the original join ordering given in the TPC-H reference outperformed the HyPer plans for Q5, Q9, Q10, and Q11 in Spark SQL, and for Q10 in Flare. For these queries, we kept the original join ordering as is. For Spark SQL, this difference is mainly due to Catalyst picking sort-merge joins over hash joins. It is worth pointing out that HyPer and Postgres plans can use indexes on primary keys, which may give an additional advantage.

Figure 10 gives the absolute execution time of Postgres, HyPer, Spark SQL, and Flare for all TPC-H queries. For all systems, data loading time is excluded, i.e., only execution time is reported. In Spark and Flare, we use `persist` to ensure that the data is loaded from memory. At first glance, the performance of Flare and HyPer lie within the same range, and notably outperform Postgres and Spark in all queries. Similarly, Spark's performance is comparable to Postgres's in most of the queries. Unlike the other systems, Postgres does not compile queries at runtime, and relies on the Volcano model [23] for query evaluation, which incurs significant overhead. Hence, we can see that Spark's query compilation does not provide a significant advantage over a standard interpreted query engines on most queries.

At a closer look, Flare outperforms Spark SQL in aggregate queries Q1 and Q6 by 32× and 13× respectively. We observe that Spark is 200× slower than Flare in nested queries (e.g., Q2) After examining the execution plans of Q2, we found that Catalyst's plan does not detect all patterns that help with avoiding re-computations, e.g., a table which has been previously scanned or sorted. In join queries, e.g., Q5, Q10, Q14, etc., Flare is faster than Spark SQL by 19×-76×. Likewise, in join variants outer join Q13, semi-join Q21, and anti-join Q22, Flare is faster by 7×, 51× and 36× respectively.

The single-core performance gap between Spark SQL and Flare is attributed to the bottlenecks identified in Sections 2.3 and 2.4. First, overhead associated with low-level data access on the JVM. Second, Spark SQL's *distributed-first* strategy that employs costly distributed operators, e.g., sort-merge join and broadcast hash join, even when running on a single core. Third, internal bottlenecks in in-memory processing, the overhead of RDD operations, and communication through Spark's runtime system. By compiling entire queries, instead of isolated query stages, Flare effectively avoids these bottlenecks.

HyPer [35] is a state-of-the-art compiled relational query engine. A precursory look shows that Flare is faster than HyPer by 10%-60% in Q4-Q5,Q7, and Q14-Q16. Moreover, Flare is faster by 2× in Q3, Q11, and Q18. On the other hand, HyPer is faster than Flare by 20%-60% in Q9, Q10, Q12, and Q21. Moreover, HyPer is faster by 2×-4× in Q2, Q8, Q17, and Q20. This performance gap is, in part, attributed to (1) HyPer's use of specialized operators like GroupJoin [32], and (2) employing indexes on primary keys as seen in Q2, Q8, etc., whereas Flare (and Spark SQL) currently does not support indexes.

In summary, while both Flare and HyPer generate native code at runtime, subtle implementation differences in query evaluation and code generation can result in faster code. For instance, HyPer uses proper decimal precision numbers, whereas Flare follows Spark in using double precision floating point values which are native to the architecture. Furthermore, HyPer generates LLVM code, whereas Flare generates C code which is compiled with GCC.

Compilation Time We compared the compilation time for each TPC-H query on Spark and Flare (results omitted). For Spark, we measured the time to generate the physical plan, which includes Java code generation and compilation. We do not quantify JVM-internal JIT compilation, as this is hard to measure, and code may be recompiled multiple times. For Flare, we measured C code generation and compilation with GCC. Both sys-

tems spend a similar amount of time on code generation and compilation. Compilation time depends on the complexity of the query, but is less than 1s for all queries, i.e., well in line with interactive, exploratory, usage.

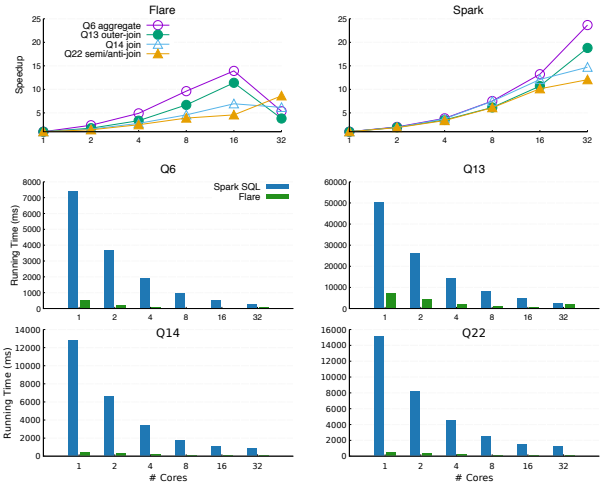


Figure 11: Scaling up Flare and Spark SQL in SF20, without NUMA optimizations: Spark has good nominal speedups (top), but Flare has better absolute running time in all configurations (bottom). For both systems, NUMA effects for 32 cores are clearly visible (Benchmark machine: 96 cores, 3 TB RAM across 4 CPU sockets, i.e., 24 cores, 750 GB each).

Parallel Scaling In this experiment, we compare the scalability of Spark SQL and Flare. The experiment focuses on the absolute performance and the Configuration that Outperforms a Single Thread (COST) metric proposed by McSherry et al. [30]. We pick four queries that represent aggregate and join variants.

Figure 11 presents speedup numbers for Q6, Q13, Q14, and Q22 when scaled up to 32 cores. At first glance, Spark appears to have good speedups in Q6 and Q13 whereas Flare's Q6 speedup drops for high core counts. However, examining the absolute running times, Flare is faster than Spark SQL by 14×. Furthermore, it takes Spark SQL estimated 16 cores in Q6 to match the performance of Flare's single core. In scaling up Q13, Flare is consistently faster by 6×-7× up to 16 cores. Similarly, Flare continues to outperform Spark by 12×-23× in Q14 and by 13×-22× in Q22.

What appears to be good scaling for Spark actually reveals that the runtime incurs significant overhead. In particular, we would expect Q6 to become memory-bound as we increase the level of parallelism. In Flare we can directly observe this effect as a sharp drop from 16 to 32 cores. Since our machine has 18 cores per socket, for 32 cores, we start accessing non-local memory (NUMA). The reason Spark scales better is because the internal overhead, which does not contribute anything to query evaluation, is trivially parallelizable and hides the memory bandwidth effects. In summary, Flare scales as expected for both of memory and CPU-bound workloads, and reflects the hardware characteristics of the workload, which means that query execution takes good advantage of the available resources – with the exception of multiple CPU sockets, a problem we address next.

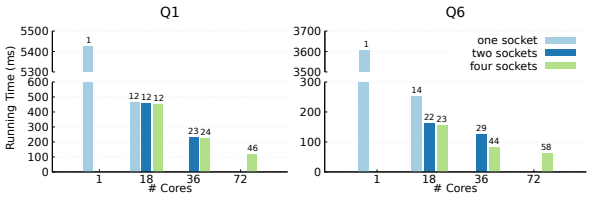


Figure 12: Scaling up Flare for SF100 with NUMA optimizations on different configurations: threads pinned to one, two, or four sockets. The speedups relative to a single thread are shown on top of the bars (Benchmark machine: 72 cores, 1 TB RAM across 4 CPU sockets, i.e., 18 cores, 250 GB each).

As a next step, we evaluate NUMA optimizations in Flare and show that these enable us to scale queries like Q6 to higher core numbers. In particular, we pin threads to individual cores and lay out memory such that most accesses are to the local memory region attached to each socket (Figure 12). Q6 performs better when the threads are dispatched on different sockets. This is due to the computation being bounded by the memory bandwidth. As such, when dividing the threads on multiple sockets, we multiply the available bandwidth proportionally. However, as Q1 is more computation bound, dispatching the threads on different sockets has little effect. For both Q1 and Q6, we see scaling up to the capacity of the machine (up to 72 cores). This is seen in a maximum speedup of 46× and 58× for Q1 and Q6, respectively.

Optimized Data Loading An often overlooked part of data processing is data loading. Flare contains an optimized implementation for both CSV files and the columnar Apache Parquet format.[2] We show loading times for each of the TPC-H tables in Table 1.

Full table read From the data in Table 1, we see that in both Spark and Flare, the Parquet file readers outperform the CSV file readers in most scenarios, despite this being a worst-case scenario for Parquet. Spark's CSV

[2] All Parquet files tested were uncompressed and encoded using PLAIN encoding.

Table	#Tuples	Postgres CSV	HyPer CSV	Spark CSV	Spark Parquet	Flare CSV	Flare Parquet
CUSTOMER	1500000	7067	1102	11664	9730	329	266
LINEITEM	59986052	377765	49408	471207	257898	11167	10668
NATION	25	1	8	106	110	< 1	< 1
ORDERS	15000000	60214	33195	85985	54124	2028	1786
PART	2000000	8807	1393	11154	7601	351	340
PARTSUPP	8000000	37408	5265	28748	17731	1164	1010
REGION	5	1	8	102	90	< 1	< 1
SUPPLIER	100000	478	66	616	522	28	16

Table 1: Loading time in ms for TPC-H SF10 in Postgres, HyPer, Flare, and SparkSQL.

reader was faster in only one case: reading `nation`, a table with only 25 rows. In all other cases, Spark's Parquet reader was 1.33×-1.81× faster. However, Flare's highly optimized CSV reader operates at a closer level of performance to the Parquet reader, with all tables except `supplier` having a benefit of less than a 1.25× speedup by using Parquet.

Performing queries Figure 13 shows speedups gained from executing queries without preloading data for both systems. Whereas reading an entire table gives Spark and Flare marginal speedups, reading just the required data gives speedups in the range of 2×-144× (Q16 remained the same) for Spark and 60%-14× for Flare. Across systems, Flare's Parquet reader demonstrated between a 2.5×-617× speedup over Spark's, and between 34×-101× over Spark's CSV reader. While the speedup over Spark lessens slightly in higher scale factors, we found that Flare's Parquet reader consistently performed on average at least one order of magnitude faster across each query, regardless of scale factor.

In nearly every case, reading from a Parquet file in Flare is approximately 2×-4× slower than in-memory processing. However, reading from a Parquet file in Spark is rarely significantly slower than in-memory processing. These results show that while reading from Parquet certainly provides performance gains for Spark when compared to reading from CSV, the overall performance bottleneck of Spark does not lie in the cost of reading from SSD compared to in-memory processing.

TensorFlow We evaluate the performance of Flare and TensorFlow integration with Spark. We run the query shown in Figure 9, which embeds a UDF that performs classification via machine learning over the data (based on a pre-trained model). As shown in Figure 14, using Flare in conjunction with TensorFlow provides speedups of over 1,000,000× when compared to PySpark, and 60× when Spark calls the TensorFlow UDF through JNI. Thus, while we can see that interfacing with an object file gives an important speed-up to Spark, the data loading ultimately becomes the bottleneck for the system. Flare, however, can optimize the data layout to reduce the amount of data copied to the bare minimum, and eliminate essentially all of the inefficiencies on the boundary between Spark and TensorFlow.

Related Work

Cluster Computing Frameworks Such frameworks typically implement a combination of parallel, distributed, relational, procedural, and MapReduce computations. The MapReduce model [18] realized in Hadoop [6] performs big data analysis on shared-nothing, potentially unreliable, machines. Twister [20] and Haloop [14] support iterative MapReduce workloads by avoiding reading unnecessary data and keeping invariant data between iterations. Likewise, Spark [55, 56] tackles the issue of data reuse among MapReduce jobs or applications by explicitly persisting intermediate results in memory. Along the same lines, the need for an expressive programming model to perform analytics on structured and semistructured data motivated Hive [52], Dremel [31], Impala [26], Shark [53] and Spark SQL [8] and many others. SnappyData [41] integrates Spark with a transactional main-memory database to realize a unified engine that supports streaming, analytics and transactions. Asterix [10], Stratosphere / Apache Flink [5], and Tupleware [17] are other systems that improve over Spark in various dimensions, including UDFs and performance, and which inspired the design of Flare. While these systems are impressive, Flare sets itself apart by accelerating actual Spark workloads instead of proposing a competing system, and by demonstrating relational performance on par with HyPer [35] on the full set of TPC-H queries. Moreover, in contrast to systems like Tupleware that mainly integrate UDFs on the LLVM level, Flare uses higher-level knowledge about specific external systems, such as TensorFlow. Similar to Tupleware, Flare's main target are small clusters of powerful machines where faults are statistically improbable.

Query Compilation Recently, code generation for SQL queries has regained momentum. Historic efforts go back all the way to System R [9]. Query compilation can be realized using code templates e.g., Spade[22] or HIQUE [27], general purpose compilers, e.g., HyPer [35] and Hekaton [19], or DSL compiler frameworks, e.g., Legobase [25], DryadLINQ [54], DBLAB [45], and LB2 [49].

Embedded DSL Frameworks and Intermediate Languages These address the compromise between productivity and performance in writing programs that can run under diverse programming models. Voodoo [39] addresses compiling portable query plans that can run

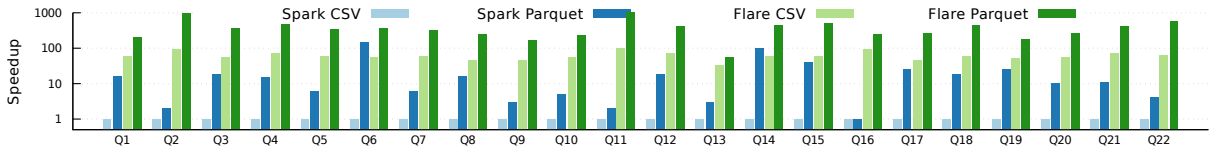


Figure 13: Speedup for TPC-H SF1 when streaming data from SSD on a single thread.

#Data Points	Spark	Spark + JNI	Flare
200	11909	990	0.064
2000	522471	3178	0.503

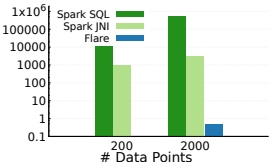


Figure 14: Running time (ms) of query in Figure 9 using TensorFlow in Spark and Flare.

on CPUs and GPUs. Voodoo's intermediate algebra is expressive and captures hardware optimizations, e.g., multicores, SIMD, etc. Furthermore, Voodoo is used as an alternative back-end for MonetDB [12]. Delite [46], a general purpose compiler framework, implements high-performance DSLs (e.g., SQL, Machine Learning, graphs and matrices), provides parallel patterns and generates code for heterogeneous targets. The Distributed Multiloop Language (DMLL) [13] provides rich collections and parallel patterns and supports big-memory NUMA machines. Weld [38] is another recent system that aims to provide a common runtime for diverse libraries e.g., SQL and machine learning. Steno [33] performs optimizations similar to DMLL to compile LINQ queries. Furthermore, Steno uses DryadLINQ [54] runtime for distributed execution. Nagel et. al. [34] generates efficient code for LINQ queries. Weld is similar to DMLL in supporting nested parallel structures.

Performance evaluation In data analytics frameworks, performance evaluation aims to identify bottlenecks and study the parameters that impact performance the most, e.g., workload, scale-up/scale-out resources, probability of faults, etc. A recent study [36] on a single Spark cluster revealed that CPU, not I/O, is the source of bottlenecks. McSherry et al. [30] proposed the COST (Configuration that Outperforms a Single Thread) metric, and showed that in many cases, single-threaded programs can outperform big data processing frameworks running on large clusters. TPC-H [51] is a decision support benchmark that consists of 22 analytical queries that address several "choke points," e.g., aggregates, large joins, arithmetic computations, etc. [11].

Conclusions

Modern data analytics need to combine multiple programming models and make efficient use of modern hardware with large memory, many cores, and NUMA capabilities. We introduce Flare: a new backend for Spark that brings relational performance on par with the best SQL engines, and also enables highly optimized heterogeneous workloads with external ML systems. Most importantly, all of this comes without giving up the expressiveness of Spark's high-level APIs. We believe that multi-stage APIs, in the spirit of DataFrames, and compiler systems like Flare, will play an increasingly important role in the future to satisfy the increasing demand for flexible and unified analytics with high efficiency.

Acknowledgments

This work was supported in part by NSF awards 1553471 and 1564207, DOE award DE-SC0018050, and a Google Faculty Research Award.

References

[1] ONNX. https://github.com/onnx/onnx.

[2] OpenMP. http://openmp.org/.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[6] Apache. Hadoop. http://hadoop.apache.org/.

[7] Apache. Parquet. https://parquet.apache.org/.

[8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in Spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[9] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: relational approach to database management. *TODS*, 1(2):97–137, 1976.

[10] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[11] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[12] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[13] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. CGO 2016, pages 194–205. ACM, 2016.

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[15] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. GPCE, pages 57–76, 2003.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[17] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[19] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[20] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *ACM HPCD*, pages 810–818. ACM, 2010.

[21] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan*, 54-C(8):721–728, 1971.

[22] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD*, pages 1123–1134. ACM, 2008.

[23] G. Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, 1994.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[25] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[26] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. a. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[27] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.

[28] J. Laskowski. Mastering Apache Spark 2. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details, 2016.

[29] F. McSherry. Scalability! but at what COST. http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html, 2015.

[30] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *HotOS*, 2015.

[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[32] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[33] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Notices*, volume 46, pages 121–131, 2011.

[34] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *VLDB*, 7(12):1095–1106, 2014.

[35] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[36] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[38] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[39] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB*, 9(14):1707–1718, 2016.

[40] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment*, 11(4), 2017.

[41] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. Snappydata: A hybrid transactional analytical store built on Spark. In *SIGMOD*, pages 2153–2156, 2016.

[42] T. Rompf and N. Amin. Functional Pearl: A SQL to C compiler in 500 lines of code. In *ICFP*, 2015.

[43] T. Rompf and M. Odersky. Lightweight Modular Staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[44] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[45] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922. ACM, 2016.

[46] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *TECS*, 13(4s):134, 2014.

[47] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[48] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[49] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. SIGMOD '18, pages 307–322. ACM, 2018.

[50] T. X. Team. XLA – TensorFlow Compiled. Post on the Google Developers Blog, 2017. http://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[51] The Transaction Processing Council. TPC-H Version 2.15.0.

[52] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[53] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[54] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *OSDI*, 8, 2008.

[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX*, HotCloud'10, 2010.

[56] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.